Technical Report
CMU/SEI-88-TR-16
ESD-88-TR-17

(2)

Carnegie-Mellon University
Software Engineering Institute

Kernel Facilities Definition

Kernel Version 3.0

Judy Bamberger
Currie Colket
Robert Firth
Daniel Klein
Roger Van Scoy

December 1989

Technical Report
CMU/SEI-88-TR-16
ESD-88-TR-17
December 1989

Kernel Facilities Definition

Kernel Version 3.0

Judy Bamberger
Currie Colket
Robert Firth
Daniel Klein
Roger Van Scoy

Distributed Ada Real-Time Kernel Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]

Karl H. Shingler
SEI Joint Program Office

Table of Contents

List of Figures

Preface

Purpose of This Document

This document defines the functionality of the Distributed Ada Real-Time Kernel (hereafter called the Kernel). The Kernel is being developed as one artifact of the Distributed Ada Real-Time Kernel Project (hereafter called the project). The goal of the Kernel is to support effectively the execution of distributed Ada applications in an embedded computer environment. As discussed in [Firth 87], the Kernel provides users with support for *language functionality* (i.e., the ability to execute Ada programs in a distributed, real-time environment); it does *not* provide support for *language features* (i.e., Ada tasking primitives). As a result, the Kernel specification places certain requirements (restrictions and conventions) on the Ada application programs that use the Kernel. These will become apparent to the reader as the definition of the Kernel is expounded.

Structure of This Document

This document is divided into three major parts:

1. Kernel Background: describes the models on which the Kernel is based and outlines the scope of its capabilities;
2. Requirements: describes the functionality and performance required of the Kernel;
3. Kernel Primitives: describes the mechanisms and primitive capabilities the Kernel provides to implement the requirements.

Kernel Facilities Definition

ABSTRACT: This document defines the conceptual design of the Kernel by specifying the underlying models, assumptions, and restrictions that govern the design and implementation of the Kernel and the behavioral and performance requirements to which the Kernel is built. This document is the requirements and top-level design document for the Kernel.

I. Kernel Background

This part of the Kernel definition provides the needed background material to understand the remainder of the document. In particular, it provides:

- The rationale for the Kernel and the goals to be achieved by it (Chapter 1).
- The definitions used in the document (Chapter 2).
- An overview of the Kernel's functionality (Chapter 3).
- A complete discussion of the assumptions, models, and restrictions on which the Kernel requirements and primitives are based (Chapter 4).

1. Rationale

Ada is now being mandated for a large number of DoD development projects as the sole programming language to be used for developing software. Many of these projects are trying to build distributed real-time systems. Many project managers and contractors are anxious to support this effort, to reap the advantages of Ada, and to use the newer techniques of software engineering that Ada can support. This transition, however, has not always been smooth; some serious problems have been encountered. This paper discusses several of these problems and describes a prototype software artifact built to address these concerns and to support execution of real-time Ada applications in a distributed, embedded environment. This prototype is not intended to solve all the problems of real-time, embedded systems, but it does provide one viable, near-term option demonstrating that Ada can be used in real-time systems today.

1.1. Ada Runtime Environment

One of the most persistent and worrying problems is the suitability of the Ada runtime system, most notably the tasking features, and especially on distributed systems. There are issues concerning functionality (amply documented in [artewg-survey 86]), customization, tool support (especially target debuggers and performance monitors); issues of inter-process communication and code distribution, and, perhaps most intractable, issues of execution-time efficiency.

One way of approaching this problem is to press for better, "more mature" Ada implementations: more optimization; user-tailorable runtime systems (as in [artewg-model 86]); special-purpose hardware. This is a valid route, but one that will take time, money, and experience, and many of the solutions will be compiler dependent, machine dependent, or application dependent. Many developers are still unsure even how to use the new language features of Ada, and at least one cycle of application use, performance measurement, and methodology review will be needed before users can be sure which parts of the Ada language and runtime are indeed critical.

The Kernel described by this document implements another route to a possible solution (defined at length in [KFD 89]) which is being pursued at the Software Engineering Institute (SEI). It should be a quicker and cheaper route, and hence a feasible short-term alternative.

1.2. Applications and Systems Code

In conventional programming, application code (which is what has to be written to meet the user requirements) is distinguished from system code (which is obtained with the target machine and which is intended to support applications generally). With Ada and embedded systems, these distinctions are not so clear cut. First, it has been traditional, when developing real-time systems in other programming languages, for the application programmer to write specific code down to a far lower level, including special device drivers, special message or signaling systems, and even a custom executive. There is far less general-purpose system code. Secondly, the Ada language complicates the distinction between application and system code. In older languages, almost all system functions were invoked through a simple and well-understood interface — the system call — expressed as a normal subroutine call. In Ada, however, many traditionally system-level functions are explicit in the language itself or implied by language constructs; for example, tasking, task communication, interrupt acquisition, and error handling. In fact, the work is really done by the old familiar system code, now disguised as the *Ada runtime*.

1.3. Abstractions and Their Breakdown

If the user is satisfied with the Ada level of abstraction — with its view of what tasks are, what time is, and so on — then the Ada view is a simplification: the application code in fact performs system calls, but the compiler inserts them automatically as part of the implementation of language constructs.

Unfortunately, many users are dissatisfied with the Ada abstraction, and seek either finer control or access to lower-level concepts, such as semaphores, send/wait or suspend/resume primitives, and bounded delays. Under the above circumstances, the extra language features, and the hidden system calls they generate, are an active hindrance to the application programmer, and an obstruction to the work of implementation.

For example, the programmer may need a strong delay primitive — one that guarantees resumption as soon as possible after the expiration of the delay. However, Ada already has a "delay" statement, but with different semantics. When implementing a different delay primitive, the user risks damaging the Ada runtime behavior, since Ada assumes it has sole control of the Ada tasks and does not expect an extra routine to perform suspensions and resumptions. To implement the new delay robustly, the user has to interface with the internals of the Ada runtime, which may be very hard to do and will surely be hard to maintain. Moreover, the Ada delay statement composes naturally into timed entry calls and timed select statements. If the user wishes to do these things with the new delay statement, a substantial part of the Ada semantics must be rebuilt, and a substantial part of the runtime must be modified.

All this, of course, is a distraction from the real work — the work of implementing the application. One of the main motivators of the Kernel is the observation that many contractors using Ada are spending most of their time worrying about the Ada system level and far too little time solving the application problems, some of which are not easy.

In sum, it can be harder to build applications using Ada language features than it would be to implement the required functionality without them. But it is also undesirable for every application to reinvent specific incarnations of real-time functional abstractions.

1.4. Distributed Applications

A further and equally difficult problem is the issue of executing applications on a distributed target configuration. Good software development methods teach decomposition of large applications into functional units communicating through well-defined interfaces. The physical allocation of such units to individual processors in the target environment can be done in many ways, without impairing their functionality. Good design therefore requires that the specification of these functional units and their interfaces be independent, as far as possible, of their physical distribution.

In a real-time system, this implies that the mechanisms by which units interact — to synchronize, communicate with, schedule, or alert one another — should be uniform, regardless of whether the units are sited on the same processor or at some distance across a distributed network. If the implementation language is Ada, this leads to a *requirement for distributed Ada.*

Unfortunately, nearly all current commercially-available Ada implementations do not support this requirement. They implement the real-time mechanisms of the language only on individual or isolated processors, and provide no help with communication between processors, and hence between units on different machines. This situation leads to systems where Ada tasks communicate by different mechanisms, with different style, semantics and implementations, merely because the Ada tasks are local in one case and remote in the other. Overall, there is a substantial loss of application clarity, maintainability, reconfigurability, and conceptual economy.

1.5. Real-Time Requirements

This brings us to the crux of the Kernel's rationale. Users — people who have to write application code — do not want language features: they want language functionality. In Ada, much of the real-time functionality is captured in the form of special features. This may well be the correct solution in the long term ([Firth 87]), since by making real-time operations explicit in the language, the compiler is permitted to apply its intelligence to their optimization and verification. But in the short term, it is palpably not working: the users either cannot use, or do not know how to use, the given features to achieve the required functionality; the implementors of the language do not know how to satisfy the variety of needs of real-time applications; the vendors are unable to customize extensively validated implementations; and commercial support for distributed targets is rare, even as the need for such support is becoming endemic among application developers.

Accordingly, it is opportune to revert to the former method of providing functionality: by specific system software implemented as a set of library routines and invoked explicitly by the user. The Kernel has taken this approach.

1.6. Purpose and Intended Audience

The main purpose of the Distributed Ada Real-Time Kernel (DARK) Project is to demonstrate that it is possible to develop application code entirely in Ada that will have acceptable quality and real-time performance. This purpose is achieved by providing a prototype artifact — a Kernel — that implements the necessary functionality required by real-time applications, but in a manner that avoids or mitigates the efficiency and maturity problems found in current Ada runtime implementations.

This prototype embodies a tool-kit approach to real-time systems, one that allows the user to build the real-time abstractions required by the application. This prototype is not intended to solve all the problems of embedded, real-time systems, nor is it the only solution to these problems. However, it is intended to be a solution where efficiency and speed are the primary motivation and, where warranted, functionality has sometimes been limited accordingly.

Given this, the purpose of such a prototype is:

1. To encourage people to use Ada for application code by mitigating many of their problems.
2. To allow developers to concentrate on the application-specific areas of their problem by providing them with a set of working system primitives that are *more familiar*, that can be invoked in a more customary way, and that can be extended.
3. To offer a usable support base, of known functionality and quality, for real-time Ada applications.

The Kernel provides one solution to the problem of using Ada in distributed, real-time, embedded applications — one that can readily be accomplished in the near term. The Kernel is truly "in the spirit of Ada" — that is, it uses the Ada language features (e.g., packages, subprograms) to provide the needed adjunct capabilities. This alternative returns explicit control of scheduling to the application implementor (as described in [Workshop 88]) and provides a uniform communication mechanism for supporting distributed systems.

2. Definitions

Two definitions are key to understanding the Kernel's models:

- *Ada task:* An Ada language construct that represents an object of concurrent execution managed by the Ada runtime environment (RTE) supplied as part of a compiler (under the rules specified in the *Ada Language Reference Manual*, see [ALRM 83]).
- *Kernel process:* An object of concurrent execution managed by the Kernel outside the knowledge and control of the Ada RTE.

The Kernel's terminology is deliberately different from that of Ada. This is for two reasons:

1. To remind the application developer to think not in Ada terms, but rather in the terms used by the Kernel.
2. To avoid the implication that the Kernel implements any specific function in a way that resembles an existing Ada feature with that function.

This document will focus specifically on *Kernel processes*, not on *Ada tasks*. In fact, except for "academic interest" or comparison purposes, the term "Ada task" does not appear in this document.

3. Kernel Functional Areas

This chapter briefly summarizes the areas that the Kernel does and does not address based on [artewg-interface 86]. First, the Kernel **does not** address the following areas:

- *Multi-level security:* This is beyond the scope of this project.
- *Rollback/checkpoint recovery:* The Kernel is not dealing with the issues of fault tolerance; however, it does address fault detection and reporting.
- *Memory/storage management and garbage collection:* In general, the Kernel expects all processes on one processor to execute in the same uniform address space, freely sharing global data. Also, the Kernel does not manage memory for access collections.
- *Pre-elaboration:* Since the Kernel is not dealing with Ada tasking (and concomittant semantics), this is not needed.
- *Fast-interrupt pragma:* The Kernel is excluding this pragma because it is specific to Ada task entries and to some compilers. The Kernel provides equivalent functionality via Kernel primitives.

The Kernel **does** address the following areas:

- *Processor management*
- *Process management*
- *Semaphore management*
- *Schedule management*
- *Communication management*
- *Interrupt management*
- *Time management*
- *Alarm management*
- *Tool interface*

These items are discussed in more detail in subsequent chapters in this document.

4. Assumptions, Models, and Restrictions

Chapter 3 defined broad, general categories of functionality. To refine these, this chapter presents a set of models, assumptions, and restrictions on which the Kernel is based.

4.1. Ada Compiler Assumptions

Regarding the Ada compiler and its relationship to the Kernel:

1. No additional pragmata are implemented or existing pragmata modified.
2. The Kernel has been developed using an Ada compiler that allows the Ada tasking RTE to be excluded from the executable image.
3. The compiler-supplied Ada runtime must be re-entrant.
4. The Ada compiler and RTE have *not* been be modified; no changes have been made that would invalidate the compiler.
5. The Kernel has total control over the system clock.

The selection of a compiler was driven by the above needs and by the needs of the model application used to test the Kernel. It was not driven by any needs of the Kernel itself. Given that no modifications have been made to the RTE, it continues to provide all services needed by the application except those related to concurrency.

4.2. Process Model

The Kernel presents to the application the abstraction of a process (as defined in Chapter 2), that is, a concurrent thread of execution. A Kernel process is a unit of code that executes in parallel with other units of code. It can communicate with other processes, can arrange to be executed at certain times, and is otherwise under the control of the Kernel's Scheduler.

In Ada terms, a process is a procedure with no parameters. The process begins execution at the start of its declarative region and ceases execution if it reaches the end of its statement sequence. This means that a process that is intended to run forever must be coded with an explicit loop statement.

A Kernel process may:

- Declare local variables
- Call Ada subprograms
- Call Kernel primitives
- Reference objects declared in packages that are Ada compilation units
- Call the Ada allocator

A Kernel process *must not*:

- Reference objects declared within other Ada subprograms (the Kernel's process encapsulation cannot set up the correct access paths to such objects).

- Be anything other than an outer-level procedure.
- Use the Ada tasking features ([ALRM 83], Chapter 9).

4.3. Initialization Model

The application image begins execution in the Main Unit (after Kernel initialization). The Main Unit is an application-supplied Ada procedure that is responsible for configuring the processor to meet the requirements of the application. This configuration must include:

1. Participating in the network initialization protocol.
2. Declaring all remote communication partners.
3. Declaring and creating all locally executing processes.

There are several optional activities that may be performed by the Main Unit, including:

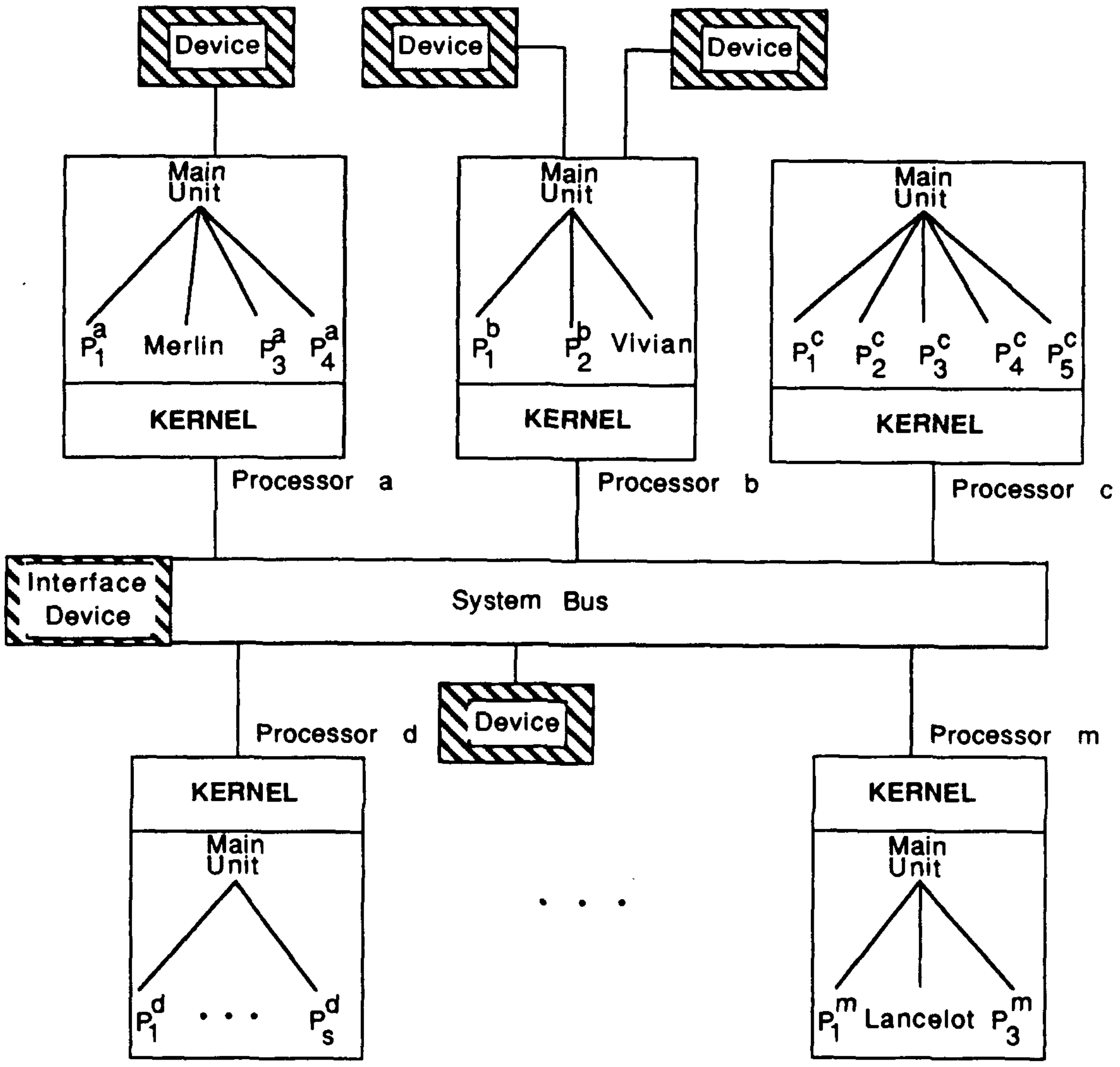
1. Allocating non-Kernel devices to processes.
2. Reading time-of-day clock (which is required for the Main Unit of the Master Processor).
3. Reporting system initialization failures to the external world.
4. Binding interrupt handlers.
5. Declaring processor-specific semaphores.
6. Performing any system-dependent initializations (devices, buses, etc.).

In general, the Main Unit is the user-supplied entity that is responsible for configuring a processor in the manner needed by the application. After initializing the processor, the Main Unit is descheduled while the processes continue to run independently.

4.4. System Model

In light of the process model discussed previously, consideration must be given to the environment in which these collections of processes are executed. This requires stepping back from the "process-in-the-small" issues and considering some system-level, or "process-in-the-large," issues. The system model on which the Kernel is based is shown in Figure 4-1. This view illustrates all the Kernel assumes about the target system:

- Three types of hardware objects in the network:
    1. Kernel processors
    2. Non-Kernel processors or devices (attached to the system bus)
    3. Devices that may interrupt a processor
- No shared memory assumed (or excluded).
- No mass storage devices assumed (or excluded).
- Kernel alone interfaces directly to the system bus.



Key

P_i^q : Process #i running on processor q.

Main Unit: The Ada Main Unit running on the processor.

Merlin, Vivian, and Lancelot are named for use in examples.

Figure 4-1: System View

Given this model, the Kernel considers that:

> The application comprises *n* Kernel processes formed into *m* Ada programs (load images) running on *m* processors.

This requires that:

- The user has a mechanism that allows for the static distribution of the *m* images over the *m* processors in the configuration.
- The application developer has a mechanism to download the images into processor memory.
- The application developer has a mechanism to commence execution of the loaded programs.
- The application developer has tools to manipulate all needed disk/tape/bulk memory accesses (if these are available in the embedded configuration).

4.5. Error Model

All Kernel requests are of the form P {S} Q; where P is the pre-condition to statement S, and Q is the post-condition of statement S. Errors occur when one of the following conditions exists:

- Pre-condition (P) is false on call, i.e., there is an error on Kernel invocation ("Invalid request").
- Pre-condition (P) is asynchronously invalidated before call terminates, i.e., an asynchronous problem arises ("Sorry, while you were waiting, something awful happened").
- The post-condition (Q) cannot be established, i.e., there is a failure of the virtual machine ("Request valid, but we can't do it today").

All Kernel errors are be signaled by a status code in all Kernel operations that may fail.

All Kernel primitives are invoked synchronously, but their return (or resumption) may be asynchronous (i.e., invoking a Kernel primitive may cause the scheduler to suspend the invoking process and resume a different process). From the point of view of the process, the entire operation appears to happen synchronously. That is, the primitives return to the caller after they have completed their operation (returning a success code) or after they have abandoned it (returning a failure code). However, their blocking behavior depends on the nature of the errors that might occur.

An error of the first. kind, where a precondition is false on call, always results in an immediate return, without blocking. An error of the second kind, where a precondition is invalidated before completion, causes a return after some interval of time, during which the caller is blocked. An error of the third kind might be detectable on call or might be detected only after some time, and so the caller might or might not have been blocked.

Wherever possible, errors are detected locally, by the Kernel on the processor running the invoking process. To do this error detection, the Kernel relies on its local copy of information representing global or remote state. A rule of this implementation is that a local copy might lag, but cannot lead, the true remote state it represents. For example, if a local process table indicates that a remote process is dead, that process has indeed died. Many of the status codes noted in this document are diagnostic in nature and thus appropriate only for software test and integration.

Given that the Kernel is intended for use in operational real-time systems, there is a means provided to disable runtime error reporting. The Kernel is configured so that each error code may be separately enabled or disabled. If an error code is enabled, the Kernel invariably checks for and reports that error. If an error code is disabled, the Kernel never reports that error, even if it occurs In addition, the Kernel is free to omit any checking whose sole purpose is to avoid damage to a process attempting an erroneous action.

However, the Kernel never omits checks that guard against damage to internal data structures or to processes other than the one performing an erroneous action, regardless of whether the error code is enabled or disabled.

4.6. Restrictions

Finally, a number of restrictions are imposed on the form of the application code. The following restrictions are offered, with justification in italics:

1. *Initialization is not a time-critical function. This is considered to be a simplification and one-time operation done at system start-up.*
2. The Kernel does not implement the Ada tasking semantics. *This is in keeping with the Kernel's design goal of making explicit control that is now implicit.*
3. No Ada tasking primitives may be used by the application. *This preserves the goal to replace the implicit operations of Ada tasking with explicit operations of Kernel processes. A second reason is to avoid having two runtime systems in the processor competing for control of the hardware clock.*
4. All Kernel processes are created statically and scheduled dynamically. *This is simply a restriction imposed to make the development effort of the Kernel a manageable activity.*
5. The Kernel does not implement fault tolerance, but it does detect and report the presence of certain (to be defined) classes of faults. *The Kernel detects certain system faults, but it leaves the recovery from these faults in the hands of the application. The Kernel provides the capability to perform fault tolerance.*
6. The Kernel does not use shared memory between nodes. *The Kernel's reliance on special hardware, such as shared memory, would restrict the portability of the Kernel and as such is disallowed.*
7. Any Kernel process may communicate with any other Kernel process. *Again, this restriction simplifies the Kernel by placing the burden of restricting communications on the system or software engineer. Management of system process names thus becomes a configuration management issue within the application.*

8. Inter-process communications are provided by explicit use of Kernel primitives. *Again, this is a manifestation of the explicit operation versus the implicit operation.*
9. Each processor has its memory completely loaded at download time. *This is another simplifying assumption for the Kernel implementation. The Kernel operates under the restriction that all processes and all data are memory resident at all times. This does not prohibit the application from building processes that can be rolled in and out of memory.*
10. The Kernel does not implement any paging or memory management facilities. *The Kernel assumes all processes on one processor execute in the same unchanging address space.*

II. Requirements

This part of the *Kernel Facilities Definition* defines the requirements to be implemented by the Kernel. The chapters in this part are parallel to those in the Kernel primitives part, which follows. Each chapter in this part is divided into two sections:

- *Behavior: Dictates the functional behavior of the Kernel.*
- Performance: Dictates the performance requirements of the Kernel. All performance requirements are based on a Motorola 68020 with a 20 Mhz clock (one wait state). This is approximately 50 machine instructions for every 12.5 μs (assuming 5 cycles/machine instruction and 20 ns/cycle).

Section numbers associated with requirements in this part are the requirement numbers referenced throughout the remainder of the document.

5. General Requirements

The requirements in this chapter apply to the specification, design, and implementation of the entire Kernel.

5.1. Behavior

5.1.1. General failure reporting

The Kernel shall return status codes to the application program as a result of an invocation of a Kernel primitive.

5.1.2. Kernel priority

Kernel primitives shall execute at the priority of the invoking process.

5.1.3. Low-level hardware details isolated from application

The Kernel shall encapsulate control and use of the system bus (i.e., the low-level communication medium).

5.1.4. System dependencies isolated

System dependencies of the Kernel shall be isolated and encapsulated within the Kernel code.

5.1.5. Kernel provably correct

The Kernel code shall be provably correct (except for termination).

5.1.6. Kernel tailoring

The Kernel shall be tailorable (via compilation) to meet the local system configuration and needs at the following levels:

1. Priority range
2. Maximum message size
3. Maximum number of process table entries
4. Maximum size of the Network Configuration Table (NCT)
5. Default timeslice quantum
6. Default message queue size

5.1.7. Kernel modularity

The Kernel shall be developed such that an application program has to import only those Kernel primitive capabilities that it needs.

5.1.8. Run-time error checking

The Kernel shall provide the capability to disable all error checking and reporting.

5.2. Performance

5.2.1. Kernel size

The Kernel shall use no more than 5% of the total available system memory for Kernel code.

5.2.2. Kernel internal data size

The Kernel shall use no more than 5% of the total available system memory for Kernel data.

5.2.3. Kernel stack for Kernel invocation

The Kernel shall use no more than 100 bytes of process stack space for a Kernel primitive invocation.

5.2.4. Linear performance

The time performance of all Kernel algorithms shall be better than linear in the number of processes.

5.2.5. Kernel algorithm documentation

All Kernel algorithms shall be fully documented in the *Kernel Architecture Manual* [KAM 89].

6. Processor Requirements

6.1. Behavior

6.1.1. Master processor identification

The Kernel shall provide the capability for exactly one processor to identify itself as the network Master processor for initialization purposes.

6.1.2. Subordinate processor identification

The Kernel shall provide the capability for all other Kernel processors to identify themselves as subordinate to the network Master processor for initialization purposes.

6.1.3. Processor failure detection

The Kernel shall detect the failure to initialize any Kernel processor in the network.

6.1.4. Processor failure reporting

The Kernel shall report to the Master processor the failure to initialize any Kernel processor in the network that is required for initialization.

6.1.5. Network clock synchronization at initialization

The Kernel shall provide the capability to synchronize at system initialization the clocks on all Kernel processors.

6.1.6. Clock synchronization *not* enforced after initialization

The Kernel shall *not* enforce clock synchronization after initial synchronization.

6.1.7. Network failure *not* detected

The Kernel shall *not* be required to detect network failure.

6.1.8. Network integrity *not* provided

The Kernel shall *not* provide network integrity controls.

6.1.9. Network integrity primitives provided

The Kernel shall provide the capability for the user to implement network integrity consistent with the needs of the application.

6.1.10. Network configuration primitives provided

The Kernel shall provide the capability for the application to specify the network configuration including:

1. Logical name
2. Physical address
3. Processor device
4. Other data required for network operation

6.1.11. Low-level network details isolated from application

The Kernel shall isolate low-level network implementation knowledge from the user application.

6.2. Performance

6.2.1. System initialization *not* time-critical

System initialization shall be accomplished in less than 5 seconds.

6.2.2. Initial time delta across the network

Immediately after system time initialization, all clocks shall be within TBD TICKs of one another.

7. Process Requirements

7.1. Behavior

7.1.1. Main Unit identification

There shall be a single Main Unit on each processor that coordinates processor-level initialization.

7.1.2. Main Unit profile

The Main Unit shall be an Ada procedure.

7.1.3. Identification of communication partners

The Kernel shall provide the capability for the Main Unit to identify all Kernel processes and non-Kernel devices with which communication from this processor is to occur.

7.1.4. Create process

The Kernel shall provide the capability for the Main Unit to create independent, concurrent threads of control (i.e., processes).

7.1.5. Ada process profile

The code of a process shall be an Ada procedure with no parameters.

7.1.6. Process stack size

The Kernel shall provide the capability for the Main Unit to specify the process stack size for each created process.

7.1.7. Process stack size fixed

The process stack size shall be fixed at process-creation time.

7.1.8. Default process stack size

The default process stack size shall be 64 bytes.

7.1.9. Message queue size

The Kernel shall provide the capability for the Main Unit to specify the incoming message queue size for each created process.

7.1.10. Message queue size fixed

The process message queue size shall be fixed at process-creation time.

7.1.11. Default message queue size

Deleted - 24 March 1989.[1]

7.1.12. Non-propagation of exceptions

The Kernel shall ensure that exceptions are not propagated outside the scope of the process in which they are raised.

7.1.13. Termination on illegal exception propagation

The Kernel shall terminate any process that attempts to propagate an exception outside the scope of the process.

7.1.14. Allocate device

The Kernel shall provide the capability for a Kernel process to identify itself as the sole receiver of messages sent from a non-Kernel device.

7.1.15. Successful initialization

The Kernel shall provide the capability for the Main Unit to inform the other Kernel processors of the successful initialization of its processor.

7.1.16. Ensure network-wide initialization completed

The Kernel shall ensure that no Kernel process executes until all necessary network-wide initializations under the control of the Kernel are complete.

7.1.17. Main Unit termination

Upon successful completion of processor initialization, the Main Unit shall be terminated.

7.1.18. Process self-termination

The Kernel shall provide the capability for a process to terminate itself in an orderly manner.

7.1.19. Process self-abortion

Deleted - 24 March 1989.[2]

[1]Message queue size is defined as the maximum number of pending messages; thus, no meaningful default exists.

[2]Subsumed by requirement 7.1.18.

7.1.20. Terminating another process

The Kernel shall provide the capability for one process to terminate another process in an orderly manner.

7.1.21. Pending messages for terminated process discarded

The Kernel shall ensure that messages pending for a terminated process are discarded.

7.1.22. Pending messages for aborted process discarded

Deleted - 24 March 1989.[3]

7.1.23. Identify self

The Kernel shall provide the capability for a process to obtain its own identity.

7.1.24. Identify another process

The Kernel shall provide the capability for a process to obtain the logical name of another Kernel process or non-Kernel device in the network.

7.1.25. Process failure detection

The Kernel shall provide the capability to detect the failure of any process in the network.

7.1.26. Process failure reporting

The Kernel shall report the detected failure of any process in the network.

7.1.27. Single message queue

The Kernel shall provide a single incoming message queue for each created process.

7.2. Performance

7.2.1. Time to create process

The creation of a process shall take no more than 60 μs.

7.2.2. Time to terminate process

The termination of a process shall take no more than 30 μs.

[3]Subsumed by requirement 7.1.21.

7.2.3. Time to abort process

Deleted - 24 March 1989.[4]

7.2.4. Time to allocate device

The allocation of a device to a process shall take no more than 20 μs.

7.2.5. Kernel data for each active process

The Kernel shall use no more than 100 bytes of data for Kernel data structures for each active process.

7.2.6. Kernel stack for each active process

The Kernel shall use no more than 64 bytes of process stack space for each process.

[4]Subsumed by requirement 7.2.2.

8. Semaphore Requirements

8.1. Behavior

8.1.1. Create a semaphore

The Kernel shall provide the capability to create a semaphore object.

8.1.2. Creation semantics

The creation of a semaphore shall associate an empty waiting process queue with the semaphore object.

8.1.3. Semaphore queue FIFO

The waiting process queue shall be first in, first out (FIFO) ordered.

8.1.4. Claiming a semaphore

The Kernel shall provide the capability for a process to obtain access to a previously created semaphore.

8.1.5. Claim timeout after duration

The Kernel shall provide the capability for a claim operation to terminate after a specified duration if the semaphore does not become available.

8.1.6. Claim timeout at specific time

The Kernel shall provide the capability for a claim operation to terminate at a specific absolute time if the semaphore does not become available.

8.1.7. Resumption priority after claim

The Kernel shall provide the capability for the claiming process to specify a priority at which it is to be unblocked.

8.1.8. Claiming an available semaphore

Claiming an available semaphore shall immediately give control of the semaphore to the invoking process and shall mark the semaphore as unavailable.

8.1.9. Claiming an unavailable semaphore

Claiming an unavailable semaphore shall immediately block the invoking process until the semaphore becomes available or the timeout expires.

8.1.10. Releasing a semaphore

The Kernel shall provide the capability for a process to relinquish control of a semaphore it currently has claimed.

8.1.11. Release semantics

Releasing a semaphore shall allow the Kernel to give control of the semaphore to the process at the head of the waiting process queue, if any.

8.1.12. FIFO exceptions

The Kernel shall allow the following exceptions to the FIFO rule for semaphore queues:

1. When a process times out on a claim, it is removed from the semaphore queue.
2. When a process is killed, it is removed from the semaphore queue.

8.1.13. Release with no waiting process

Releasing a semaphore shall allow the Kernel to mark that semaphore as free if there is no process waiting for that semaphore.

8.1.14. Created semaphore is free

When a semaphore is created, it shall be marked as free (i.e., available).

8.1.15. Release in any order

The Kernel shall allow a process to release claimed semaphores in any order.

8.1.16. Claim with zero-length timeout

Claiming a semaphore with a zero-length timeout[5] shall immediately: claim the specified semaphore or terminate the claim operation.

8.2. Performance

8.2.1. Time to create a semaphore

Creating a semaphore object shall take no more than 25 μs.

[5]A zero-length timeout is a duration of zero or less, an absolute time of now or in the past, or any time reference shorter than that required for the Kernel to perform a context switch.

8.2.2. Time to claim a semaphore

Claiming a semaphore object shall take no more than 25 μs.[6]

8.2.3. Time to release a semaphore

Releasing a semaphore object shall take no more than 25 μs.[7]

[6]No scheduling activity is involved.

[7]No scheduling activity is involved.

9. Scheduling Requirements

9.1. Behavior

9.1.1. Initial scheduling parameters

The Kernel shall provide the capability for each Main Unit to specify the initial scheduling parameters of each process to execute on that processor:

9.1.2. Default schedule parameters

The default schedule parameters of a process shall be: preemptable and lowest possible priority.

9.1.3. Representation of priority

Process priorities shall be represented as a subset of the positive integers.

9.1.4. Definition of priority range

The range of priorities shall be defined at compile time by the user(s) of the Kernel.

9.1.5. Definition of priority ordering

Smaller integer values for priority shall represent higher process priorities.

9.1.6. Dynamic process priority

The Kernel shall provide the capability to set process priorities dynamically.

9.1.7. Set initial priority

The Kernel shall provide the capability to set the initial process priority at process-creation time.

9.1.8. Change priority

The Kernel shall provide the capability for a process to change its own process priority.

9.1.9. Query priority

The Kernel shall provide the capability for a process to query its own process priority.

9.1.10. Definition of preemption

The Kernel shall provide two preemption states: preemptable by a process of the same priority, and not preemptable by a process of the same priority.

9.1.11. Set initial preemption

The Kernel shall provide the capability to set the initial preemption state at process-creation time.

9.1.12. Change preemption

The Kernel shall provide the capability for a process to set its own preemption state.

9.1.13. Query preemption

The Kernel shall provide the capability for a process to query its own preemption state.

9.1.14. Definition of wait

The Kernel shall provide the capability for a process to voluntarily relinquish control of the processor and to be resumed at the next sequential statement.

9.1.15. Wait for duration

The Kernel shall provide the capability for a process to block its own execution for a specified duration.

9.1.16. Wait until specified time

The Kernel shall provide the capability for a process to block its own execution until a specified time.

9.1.17. Resumption priority after wait

The Kernel shall provide the capability for a waiting process to specify a priority at which it is to be unblocked.

9.1.18. Set timeslice

The Kernel shall provide the capability to set the timeslice quantum for each processor.

9.1.19. Enable round-robin time slicing

The Kernel shall provide the capability to enable the time slicing of processes of equal priorities in a round-robin manner.

9.1.20. Disable round-robin time slicing

The Kernel shall provide the capability to disable time slicing of processes of equal priorities in a round-robin manner.

9.1.21. Default timeslice settings

The default timeslice settings shall be: time slicing disabled and timeslice quantum set to 1 second.

9.1.22. Setting another process's scheduling parameters prohibited

The Kernel shall prohibit one process from directly setting another's scheduling parameters (e.g., priority, preemption) with the exception of the initial scheduling parameters, which are defined by the Main Unit at process-creation time.

9.1.23. Process states

A process shall always be in one of four definite states:

1. Running
2. Dead
3. Blocked
4. Suspended

9.1.24. Scheduling algorithms

The Kernel scheduling algorithms shall be:

1. Deterministic, and
2. *Of predictable performance.*

9.1.25. Scheduling algorithms provided

The scheduling algorithms shall provide priority-based, preemptive scheduling.

9.1.26. Documentation of scheduling algorithms

The Kernel scheduling algorithms shall be fully documented in the *Kernel User's Manual.*

9.1.27. Meaning of a timeslice

A process that is preemptable while time slicing is enabled shall execute for no more than one timeslice quantum before a forced reschedule point occurs.

9.1.28. Priorities incommensurable across processors

The Kernel shall assume that process priorities are incommensurable across processors.

9.1.29. Zero-length waits

The Kernel shall process any zero-length wait[8] as a wait that has expired immediately.

[8]A zero-length wait is a duration of zero or less, an absolute time of now or in the past, or any time reference shorter than that required for the Kernel to perform a context switch.

9.2. Performance

9.2.1. Time to set priority

Setting a process priority shall take no more than 57 μs.[9]

9.2.2. Time to set preemption

Setting a process preemption state shall take no more than 18 μs.[10]

9.2.3. Time to suspend process

Suspension of a process shall take no more than 43 μs.

9.2.4. Time to resume process

Resumption of a process shall take no more than 33 μs.

9.2.5. Time to enable or disable time slicing

Enabling or disabling timeslice scheduling shall take no more than 18 μs.[11]

9.2.6. Time of context switch

The time needed to suspend execution of the currently running process and resume *execution of a different process* shall take no more than 76 μs.[12]

9.2.7. Dispatch time

The Kernel shall take no more than 33 μs to dispatch the highest priority task after it becomes unblocked.

[9]No scheduling activity is involved.

[10]No scheduling activity is involved.

[11]No scheduling activity is involved.

[12]*Requirement 9.2.3 time + requirement 9.2.4 time.*

10. Communication Requirements

10.1. Behavior

10.1.1. Reference communication partners

The Kernel shall provide the capability for a Kernel process to reference all other Kernel processes and non-Kernel devices with which it communicates.

10.1.2. Send

The Kernel shall provide the capability for a Kernel process to send data to another Kernel process or non-Kernel device.

10.1.3. Send with ACK

The Kernel shall provide the capability for the sending Kernel process to request acknowledgment (ACK) of message receipt by the receiving Kernel process.

10.1.4. Resumption priority for send with ACK

The Kernel shall provide the capability for the sending Kernel process to specify the priority at which it is to be unblocked when it requests an acknowledgment.

10.1.5. Sender specifies recipient

The Kernel process originating a message sent to a single recipient shall specify the identity of the receiving Kernel process or non-Kernel device.

10.1.6. Receiver physical address hidden

The Kernel process that sends a mesage shall not need to know the physical network address of the Kernel process or non-Kernel device to which the message is to be sent.

10.1.7. Send timeout after specified duration

The Kernel shall provide the capability for a Kernel process to terminate a send with ACK operation if the message is not received after a specified duration.

10.1.8. Send timeout by specified time

The Kernel shall provide the capability for a Kernel process to terminate a send with ACK operation if the message is not received by a specified time.

10.1.9. Send timeout reporting

The Kernel shall terminate with a status code any send operation whose timeout expires.

10.1.10. Send recipient failure detection

The Kernel shall detect the failure of the Kernel process that is to receive a message via a send with ACK operation.

10.1.11. Send recipient failure reporting

The Kernel shall report the failure of the Kernel process that is to receive a message during a send with ACK operation.

10.1.12. Receive

The Kernel shall provide the capability for a Kernel process to receive data from another Kernel process or non-Kernel device.

10.1.13. Resumption priority for receive

The Kernel shall provide the capability for the receiving Kernel process to specify the priority at which it is to be unblocked.

10.1.14. Kernel identifies sender

The Kernel shall inform the Kernel process that receives the message of the identity of the sending Kernel process.

10.1.15. Sender physical address hidden

The Kernel process that receives a message shall not need to know the physical network address of the Kernel process or non-Kernel device that sent the message.

10.1.16. Receive timeout after specified duration

The Kernel shall provide the capability for a Kernel process to terminate a receive operation if it is not completed after a specified duration.

10.1.17. Receive timeout by specified time

The Kernel shall provide the capability for a Kernel process to terminate a receive operation if it is not completed by a specified time.

10.1.18. Receive timeout reporting

The Kernel shall terminate with a status code a receive operation whose timeout expires.

10.1.19. Message format

The Kernel shall not impose any restrictions on the content, format, or length of a message other than maximum message size limitations.

10.1.20. Message too big on receive

Any attempt to receive a message larger than the work space provided by the receiving Kernel process shall result in the message contents being discarded, but the message attributes shall be delivered to the receiving Kernel process. These include:

1. Message sender's process identifier
2. Message tag[13]
3. Message length

10.1.21. FIFO message queue

The Kernel process incoming message queue shall be FIFO ordered.

10.1.22. Message queue overflow

The Kernel shall provide two options for managing the incoming message queue when the arrival of a new message would cause queue overflow:

1. Accept the new message by overwriting old messages; or
2. Reject the new message.

10.1.23. Default overflow handling

The default option for managing incoming message queue overflow shall be to reject the new message.

10.1.24. Status code on queue overflow

The Kernel shall return a status code on the first receive operation after incoming message queue overflow has occurred.

10.1.25. Communication failure reporting

When the Kernel detects that a Kernel process has failed, it shall terminate pending communication operations with that process and return an appropriate status code.

10.1.26. Communication with non-Kernel devices

The Kernel shall provide the capability to communicate with any device (capable of sending or receiving communication) in the network.

10.1.27. Communication deadlock detection

The Kernel shall prohibit a process from performing a send with ACK operation to itself.

[13]No tag is received for messages from non-Kernel devices.

10.1.28. Common primitives for all communication

The Kernel shall provide a common set of communication primitives for both local and remote message passing.

10.1.29. Local communication optimization

The Kernel shall optimize communications local within a processor.

10.1.30. Message integrity

The Kernel shall ensure message integrity, including:

1. The entire message reaches the receiver; and
2. Simple transmission errors are detected.

10.1.31. Communication integrity

The Kernel shall ensure communication integrity, including:

1. The sender of a message has been identified to the Kernel as a Kernel process or a non-Kernel device.
2. The receiver of a message has been identified to the Kernel as a Kernel process or a non-Kernel device.

10.1.32. Send with ACK blocks

The sending Kernel process shall block immediately, and shall remain blocked, until the requested acknowledgement is received or the timeout expires.

10.1.33. Message Acknowledgment

When a send with ACK is performed, the receiving Kernel shall acknowledge receipt of the message after the message contents have been copied completely into the work space provided by the receiving Kernel process.

10.1.34. Acknowledgments handled automatically

When a send with ACK is performed, the receiving Kernel shall automatically inform the sending Kernel of the success or failure of message receipt.

10.1.35. Receive potentially blocking

The receiving Kernel process shall block immediately, and shall remain blocked, until a message is received or the timeout expires.

10.1.36. Message too big reporting

The Kernel shall report the occurence of receiving a message larger than the work space provided by the receiving Kernel process.

10.1.37. Send with ACK timeout

The timeout specified by the send with ACK capability shall be executed on the processor of the receiving Kernel process.

10.1.38. Receive with a zero-length timeout

Receiving a message with a zero-length timeout[14] shall immediately: cause the pending message (if any) to be copied into the work space of the receiving Kernel process or indicate that no message was pending.

10.1.39. Send with ACK with zero-length timeout

If the timeout specified by the send with ACK capability is a zero-length timeout[14] and the receiving Kernel process is not pending on a receive, the Kernel shall immediately return a negative acknowledgment to the sending Kernel.

10.2. Performance

10.2.1. Transmission time of 0-length message

The time[15] needed to send a 0-length message without acknowledgment to a single process in the case of two processes on two processors (excluding network transmission time) shall be no more than 25 μs.

10.2.2. Transmission time of large message

The time needed to send a large[16]message without acknowledgment to a single process in the case of two processes on two processors (excluding network transmission time) shall be no more than 25 μs.

10.2.3. Transmission time of 0-length message with ACK

The time needed to send a 0-length message with acknowledgment to a single process waiting for that message in the case of two processes on two processors (excluding network transmission time) shall be no more than 50 μs.

[14]A zero-length wait is a duration of zero or less, an absolute time of now or in the past, or any time reference shorter than that required for the Kernel to perform a context switch.

[15] All times exclude message copy time and net transmission time.

[16] The actual size of a "large" message is intentionally unspecified. The intent is to select a message size that places a non-trivial load on the system.

10.2.4. Transmission time of large message with ACK

The time needed to send a large message with acknowledgment to a single process waiting for that message in the case of two processes on two processors (excluding network transmission time) shall be no more than 50 μs.

10.2.5. Transmission time of 0-length message locally

The time needed to send a 0-length message without acknowledgment to a single process in the case of two processes on the same processor shall be no more than 15 μs.

10.2.6. Transmission time of large message locally

The time needed to send a large message without acknowledgment to a single process in the case of two processes on the same processor shall be no more than 15 μs.

10.2.7. Transmission time of 0-length message locally with ACK

The time needed to send a 0-length message with acknowledgment to a single process waiting for that message in the case of two processes on the same processor shall be no more than 20 μs.

10.2.8. Transmission time of large message locally with ACK

The time needed to send a large message with acknowledgment to a single process waiting for that message in the case of two processes on the same processor shall be no more than 20 μs.

10.2.9. Time to receive 0-length message

The Kernel overhead time needed to receive a 0-length available message shall be no more than 25 μs.

10.2.10. Time to receive large message

The Kernel overhead time needed to receive a large available message shall be no more than 25 μs.

10.2.11. Fixed message overhead

The Kernel shall use no more than 128 bits/message.

11. Interrupt Requirements

11.1. Behavior

11.1.1. Interrupt names

The Kernel shall provide the abstraction of an **Interrupt name** that shall be used in all Kernel operations involving interrupts.

11.1.2. No exception propagation from interrupt handlers

No exceptions shall be propagated out of the scope of an interrupt handler.

11.1.3. Exit on exception propagation from interrupt handlers

Any attempt to propagate an exception outside the scope of an interrupt handler shall result in the immediate return from the interrupt handler.

11.1.4. Enable interrupt

The Kernel shall provide the capability for a process to enable any available interrupt.

11.1.5. Disable interrupt

The Kernel shall provide the capability for a process to disable any available interrupt.

11.1.6. Default status of interrupts

All interrupts available to the application code shall be disabled and unbound by default.

11.1.7. Query interrupt status

The Kernel shall provide the capability for a process to query the enabled/disabled status of any available interrupt.

11.1.8. Simulate interrupt

The Kernel shall provide the capability for a process to simulate any available interrupt in software.

11.1.9. Define interrupt handler in Ada

The Kernel shall provide the capability for an interrupt handler to be defined in Ada.

11.1.10. Ada interrupt handler profile

The Ada interrupt handler shall be a procedure with no parameters.

11.1.11. Bind interrupt handler

The Kernel shall provide the capability to define an Ada code unit as an interrupt handler for any available interrupt.

11.1.12. Blocking prohibited in interrupt handler

Within an interrupt handler, any attempt to invoke a blocking Kernel primitive in a situation where it would block shall be terminated immediately with a status code.

11.1.13. Interrupts not queued

The Kernel shall not queue pending interrupts.

11.1.14. Interrupt implementation visibility

The Kernel shall allow the user unhindered access to the underlying hardware implementation of interrupts.

11.1.15. Interrupt priority

All interrupt handlers shall execute at a higher priority than any Kernel process.

11.1.16. Reserved interrupts

The Kernel shall reserve the interrupts known to it on the target hardware that it needs to function, and those that it knows are needed by the Ada runtime.

11.1.17. Available interrupts

The Kernel shall make available to the application all other interrupts known to it on the target hardware.

11.1.18. State not relevant for interrupt handler

Within an interrupt handler, any attempt to modify the state of a process shall be rejected immediately with a status code.

11.1.19. Process attributes not relevant for interrupt handler

Within an interrupt handler, any attempt to modify or query process attributes shall be rejected immediately with a status code.

11.1.20. Interrupt name details hidden

The Kernel shall encapsulate the implementation details associated with the Interrupt name abstraction.

11.1.21. Interrupt name visibility

The Kernel shall provide an **interrupt name** for each interrupt known to it on the target hardware.

11.2. Performance

11.2.1. Time to enter interrupt handler

The time needed to enter an interrupt handler (from moment of preemption until the first statement of the handler is executed) shall be less than 15 μs.[17]

11.2.2. Time to exit interrupt handler

The time needed to exit an interrupt handler (from the return from interrupt until the resumption of the preempted process) shall be less than 10 μs.

11.2.3. Time to bind interrupt handler

The time needed to bind an interrupt handler shall be less than 20 μs.

11.2.4. Interrupt stack use

The Kernel shall use no more than 16 bytes[18] of process stack space per interrupt handler invocation.

[17]No context save is included in this time.

[18]No context data are included in this allocation.

12. Time Requirements

12.1. Behavior

12.1.1. Package Calendar

The Kernel shall support the standard Ada package Calendar (see [ALRM 83], Section 9.6).

12.1.2. Exclusion of Package Calendar

The Kernel shall not require that package Calendar be a part of the load image.

12.1.3. Use of real-time clock

The Kernel shall be capable of using a real-time clock.

12.1.4. Definition of TICK

The Kernel shall export a constant, TICK, that gives the maximum granularity of time measurement as an integral number of microseconds.

12.1.5. Definition of SLICE

The Kernel shall export a constant, SLICE, that represents the minimum usefully schedulable unit of time.

12.1.6. Relationship between TICK and SLICE

The value of SLICE shall be a multiple of TICK.

12.1.7. TICK used internally

The Kernel shall maintain local elapsed time accurate to within one TICK.

12.1.8. SLICE basis for scheduling

The Kernel shall maintain all scheduling event times accurate to at least one SLICE.

12.1.9. External time representation

The Kernel shall make two kinds of time abstractions available to the application:

1. An elapsed time (a relative time, similar to the Ada type *duration*); and
2. An epoch time (an absolute time, similar to the Ada type *time*).

12.1.10. Adjust local processor elapsed time

The Kernel shall provide the capability to adjust (increment and decrement) local processor elapsed time.

12.1.11. Effect of adjusting local processor elapsed time

Adjusting local processor elapsed time shall affect all pending events (i.e., those pending for an elapsed time and those pending until an epoch time).

12.1.12. Reset local processor epoch time

The Kernel shall provide the capability to reset the local processor epoch time.

12.1.13. Effect of resetting local processor epoch time

Resetting local processor epoch time shall affect only those events pending until an epoch time.

12.1.14. Time base reference

The time base reference shall be Julian day 1 (i.e., 1200 on 1 January 4713BCE; see Joseph Justus Scaliger, *De emendatione temporum*, 1582).

12.1.15. Read clock

The Kernel shall provide the capability to read current epoch time.

12.1.16. Clock synchronization

The Kernel shall provide the capability to synchronize to the time of the invoking processor the clocks on all other Kernel processors.

12.1.17. Definition of synchronization

Synchronization of network clocks shall force:

1. All epoch times to be set to the epoch time on the invoking processor.
2. All elapsed times to be identical to the elapsed time on the invoking processor, within the delta defined in 12.2.5.

12.1.18. Synchronize timeout after duration

The Kernel shall provide the capability for a synchronize operation to terminate after a specified duration if the synchronization protocol has not successfully completed.

12.1.19. Synchronize timeout at specific time

The Kernel shall provide the capability for a synchronize operation to terminate at a specific absolute time if the synchronization protocol has not successfully completed.

12.1.20. Resumption priority after synchronize

The Kernel shall provide the capability for the synchronizing process to specify a priority at which it is to be unblocked.

12.1.21. Duplicate synchronization rejected

The Kernel shall reject an invocation of the sychronize primitive while a previous invocation is in progress.

12.1.22. Synchronize potentially blocking

The synchronizing process shall block immediately, and shall remain blocked, until the synchronization is complete or the timeout expires.

12.1.23. Arithmetic operations defined for time

The Kernel shall define arithmetic and comparison operations on both kinds of time and conversions between them and Ada *duration*.

12.1.24. Arithmetic exceptions defined for time

The Kernel shall detect range violations caused by arithmetic or conversion operations on time types, and shall raise the appropriate predefined exceptions.

12.1.25. Time representation

The Kernel shall represent both epoch and elapsed time in units of one microsecond.

12.1.26. Scheduling interval

The Kernel shall not increase scheduling time intervals specified by the application; it may, however, decrease them to the next lower multiple of SLICE.

12.1.27. Synchronize with zero-length timeout

Synchronize with a zero-length timeout[19]shall immediately cause: the synchronize operation to commence or return a status code.

12.2. Performance

12.2.1. Time to adjust local processor time

The time needed to adjust the local processor time shall be no more than 18 μs.[20]

[19]A zero-length wait is a duration of zero or less, an absolute time of now or in the past, or any time reference shorter than that required for the Kernel to perform a context switch.

[20]No scheduling activity is involved.

12.2.2. Time to reset epoch time

The time needed to reset the epoch time shall be no more than 18 μs.[21]

12.2.3. Time to read clock

The time needed to read the clock shall be no more than 18 μs.

12.2.4. Time to synchronize

The time needed to synchronize all local processor clocks shall be no more than 200 μs.[22]

12.2.5. Accuracy of time synchronization

After synchronization all processor clocks shall be within TBD μs.

[21] No scheduling activity is involved.

[22] This is 4 * (time needed to send a 0-length message); see requirement 10.2.3 in Chapter 10.

13. Alarm Requirements

13.1. Behavior

13.1.1. Maximum number of alarms

The user shall be able to define one alarm per process.

13.1.2. Relative alarm time

The Kernel shall provide the capability to set an alarm to expire after a specified duration.

13.1.3. Absolute alarm time

The Kernel shall provide the capability to set an alarm to expire at a specified time.

13.1.4. Kernel defined alarm exception

The Kernel shall define an **alarm_expired** exception.

13.1.5. Expiration of alarm

The expiration of an alarm shall raise the **alarm_expired** exception in the process that set the alarm.

13.1.6. Transfer priority

When a process sets an alarm, it shall have the capability to specify a priority at which it is to execute when the alarm expires and the process is eligible to run.

13.1.7. Set alarm for zero seconds

Setting an alarm to expire in zero time units shall immediately raise the **alarm_expired** exception.

13.1.8. Set alarm for non-future duration

Setting an alarm to expire after some non-future duration shall immediately raise the **alarm_expired** exception.

13.1.9. Set alarm for time in past

Setting an alarm to expire at some time in the past shall immediately raise the **alarm_expired** exception.

13.1.10. Cancel alarm

The Kernel shall provide the capability to cancel an unexpired alarm.

13.1.11. Expiration of alarm cancels pending events

The expiration of an alarm shall cancel any other pending event for this process.

13.2. Performance

13.2.1. Time to set alarm

Setting an alarm shall take no more than 10 μs.

13.2.2. Time to cancel alarm

Canceling an alarm shall take no more than 10 μs.

13.2.3. Time to transfer to exception handler

The time needed to raise the **alarm_expired** exception and transfer control to the inner-most enclosing exception handler shall take no more than TBD[23] μs.

[23]This is compiler-dependent and not under the control of the Kernel.

14. Tool Interface

The requirements in this chapter are not mapped on the Kernel primitives in Chapters 15 - 22, nor are the tool interface primitives mapped onto the other requirements. The tool interface is a non-essential part of the Kernel, and its existence is not needed for the proper use and functioning of the Kernel.

14.1. Behavior

14.1.1. Monitoring

The Kernel shall provide the capability to monitor certain internal data and primitive operations.

14.1.2. Monitor process

The monitor shall be a process.

14.1.3. Number of monitors

The application may have any number of monitor processes within the application-defined constraints.

14.1.4. Monitor process is local

The monitor process shall be local to the processor it is monitoring.

14.1.5. Asynchronous logging

The Kernel shall log the monitored data by sending a message to the local monitor process.

14.1.6. Monitor process attributes

The Kernel shall provide the capability to monitor process attributes of any declared process.

14.1.7. Specify processes to monitor

The Kernel shall provide the capability to specify the processes on which monitoring is to occur.

14.1.8. Disable monitoring of process attributes

The Kernel shall provide the capability to disable monitoring process attributes.

14.1.9. Process attributes available

The process attributes that shall be available for monitoring are:

1. Process identifier
2. Process state
3. Time process entered state
4. Process priority
5. Process preemption state
6. Process alarm state
7. Return status code (if the state changes as a result of an invocation of a Kernel primitive)

14.1.10. Monitor message attributes

The Kernel shall provide the capability to monitor message attributes.

14.1.11. Disable monitoring of message attributes

The Kernel shall provide the capability to disable monitoring message attributes.

14.1.12. Message attributes available

The message attributes that shall be available for monitoring are:

1. Sender's process identifier
2. Receiver's process identifier
3. Time message was sent or received
4. Message tag
5. Message length

14.1.13. Monitor message contents

The Kernel shall provide the capability to monitor message contents.

14.1.14. Disable monitoring of message contents

The Kernel shall provide the capability to disable monitoring message contents.

14.1.15. Process table available

The Kernel shall provide the capability to read the Kernel's process table, which includes:

1. Process name
2. Process identifier
3. Process state
4. Processor address
5. Size of process table

14.1.16. Interrupt table available

The Kernel shall provide the capability to read the Kernel's interrupt table, which includes:

1. Interrupt
2. State
3. Interrupt condition

14.1.17. No communication overhead incurred

The Kernel shall not perform any background Kernel-to-Kernel communication to collect process, message, or interrupt attributes.

14.1.18. Monitor requests ignored

The Kernel shall ignore all requests to monitor processes that are unknown, aborted, or terminated when monitoring is enabled.

14.2. Performance

14.2.1. Time to define a monitoring activity

The definition of a monitoring activity shall take no more than 10 μs.

14.2.2. Time to terminate a monitoring activity

The termination of a monitoring activity shall take no more than 5 μs.

14.2.3. Time to process a monitoring activity

The processing of a monitoring activity shall take no more than 5 μs.

14.2.4. Tool predictability

The tool interface primitives shall be predictable in their use of time and memory resources.

14.2.5. Time measured consistently

The Kernel shall perform all time measurements at a consistent point in the Kernel code.

14.2.6. Data logged consistently

The Kernel shall send all logging messages at a consistent point in the Kernel code.

III. Kernel Primitives

The logical result of the models, assumptions, and restrictions in Chapter 4 is a Kernel familiar to most embedded systems software engineers. The Kernel combines many of the known and proven forms from traditional (i.e., non-Ada) systems with some of the desired "extensions" to Ada (based, in part, on the thinking embodied in the [artewg-interface 86] report), resulting in a software artifact that can be ported and used by Ada applications. These capabilities appear to the application program as a collection of Ada packages—reusable components—that, together with certain application programming conventions, can be combined with the Ada application to execute in a distributed, hard real-time, embedded environment.

The basic format of each chapter is:

- General discussion of implications the Kernel models on the primitives.
- The Kernel primitives, their functionality and status codes.
- Blocking conditions, where appropriate.
- Status codes and their explanations (when needed).

The Kernel communication model presents a set of primitives to the application, and implements those primitives on an underlying set of distributed processors connected by data paths. The model, the implementation, and the intended mode of use, can all be related to the International Standards Organization (ISO) Reference Model (see [Zimmermann 80] and [Tanenbaum 81]), which provides a conceptual framework for organizing the Kernel primitives, as shown in Figure 14-1. The ISO Reference Model identifies seven layers, named, from lowest to highest:

1. Physical
2. Data Link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

The target hardware provides Layer 1. The Kernel implements Layers 2 to 4, and therefore presents to the application the Transport layer. The Kernel thus encapsulates within itself the Data Link and Network layers, rendering them invisible to the application. The application code can implement Layers 5 to 7, in part by using other Kernel primitives.

Layer		Kernel Equivalent
7	Application	Created by user
6	Presentation	Created by user
5	Session	Kernel primitives: declare process, create process, allocate device receiver, and initialization complete
4	Transport	Kernel primitives: send message, send message and wait, and receive message
3	Network	Null (can be built by user)
2	Data Link	Datagram model
1	Physical	Built by using Kernel primitives: network configuration table, initialize master processor, and initialize subordinate processor

Figure 14-1: ISO Model to Kernel Mapping

Physical Layer

The Physical layer is represented by the hardware data paths, which support the transmission of a serial bitstream between processors. These hardware data paths are used by the Kernel in a *packet switching* mode; that is, a sequence of bits—a *frame*—is sent at the discretion of the originator, with no implied reservation of resources or preservation of state between frames.

Data Link Layer

This is the layer at which basic error detection and recovery and flow control may be provided. The Kernel uses a simple *datagram* model, in which a frame is transmitted with no acknowledgment, no error correction, and no flow control. Minimal error detection is achieved by using a datagram checksum,[24] but any recovery is performed by application code (i.e., above the Transport layer). Similarly, datagram storage overflow is recognized and reported at the Transport layer.

Network Layer

Currently, the Kernel has a null Network layer. The Kernel assumes that point-to-point communication is available between any pair of nodes (processors). Routing is accomplished trivially in the sender by dispatching a point-to-point datagram directly to the receiver; no alternative routing is provided.

However, since the abstraction presented by the Kernel to the application is above this layer, a real Network layer could subsequently be added without requiring any application code to be changed.

Transport Layer

The Kernel builds the Transport layer by using the network connections identified by Kernel primitives and data structures at the Physical and Session layers.

The physical network is described by a Network Configuration Table (shown in Figure 15-1), a copy of which is maintained in each processor. This table is created by the application developer and is communicated to the Kernel during application initialization. Once that information is provided, the Kernel verifies the network connectivity and opens the physical connections between processors.

Subsequently, the logical *processes* and their physical *sites* are communicated to the Kernel. The model on which the Kernel is based assumes that all processes are created at initialization time, that a process never moves, and that a process once dead is never restarted. The Kernel therefore computes the logical-to-physical mapping once only and never subsequently changes it. Attempts to communicate with dead processes are treated as transport errors.

[24]Null in the current implementation.

The Transport layer also performs the conversions between *messages* and the underlying datagrams. Currently, this is done trivially by using one datagram per message or per acknowledgment, and if necessary by restricting the maximum message size accordingly.

The Transport layer is the layer visible to the application. It supports unacknowledged send operations and end-to-end acknowledged send operations. All errors detected in this or any lower layer are reported at this layer, in the form of status codes returned by the Kernel primitives.

Session Layer

This layer is implemented by application code. Since it establishes logical connections between processes, its presence is required, and the application developer must write specific code to create it. This code is part of the application initialization code; it must be present on every processor and, in Ada terms, must be part of the Main Unit on that processor.

The model is one of a set of logical processes, each with an application-defined *logical name* and each with a single *incoming message queue* for the reception of messages from other processes.

The Kernel primitive *declare_process* indicates an intent to create or communicate with a given named process. It establishes the mapping between application-level process names and Kernel internal names.

The Kernel primitive *create_process* creates the process environment, establishes its incoming message queue, allocates stack resources, and makes that queue available to the network. Thereafter, one process may communicate with another.

Presentation Layer

In the Kernel model, the Presentation layer performs no transformation of data. Rather, it performs the translation between Ada values — values of application-defined data types — and message values. This is done by application code. The purpose of the Presentation layer is to establish above the Transport layer the strong typing of the Ada language, by ensuring that communicating processes pass only strongly typed data and do so by referencing a common set of data conversion routines bound to a common Ada data type.

Application Layer

This layer uses the Presentation layer for whatever purpose the code requires. The model here is of parallel independent threads of control executing Ada code, identifying each other by application-level symbolic names, and communicating by passing values of Ada data types.

15. Processor Management

There are two steps to using the system model shown in Figure 4-1. Note that the initialization of the system topology has been deliberately kept simple. This facilitates the development of the Kernel, keeps the initialization interfaces simple, and allows the users of the Kernel to develop more readily their own system-specific initialization software. First, the physical topology of the system must be defined; secondly, the system must be initialized. The approach taken to achieve the first step requires that the application engineer first define the network configuration in a manner that the Kernel understands. This is done using the Network Configuration Table (NCT) shown in Figure 15-1.

Logical Name	Physical Address	Kernel Device	Needed To Run	Initialization Order

Figure 15-1: Sample Network Configuration Table (NCT)

The NCT provides the minimum information needed by the Kernel to perform system initialization and its inter-process communication functions. It is supplied by the application to the Kernel; it is implementation and hardware dependent and is available to the application for implementation of higher levels of network integrity. For each device accessible over the network, this table defines the following information:

- Logical name: Logical (string-valued) name for the device.
- Physical address: Hardware-specific information needed to access the device over the system bus.
- Kernel device: Identifies those devices that execute the Kernel. It is possible to communicate with non-Kernel devices, so they must understand enough of the communication protocols to make this possible. But they are not expected to participate in the network initialization protocol. Non-Kernel devices place the burden of initialization and message formatting upon the application. That is, the Kernel routes messages to and receives messages from non-Kernel devices, but it is the responsibility of the application to format and unformat these messages.
- Needed to run: Identifies those devices that must be available at initialization time in order for the application to begin execution. This could be used to mark failed or spare devices at startup.
- Initialization order: Identifies the order in which the Kernel nodes of the network are to be initialized. The default, unless specifically overridden, is for the nodes to initialize in the order in which their entries occur in the NCT.

To achieve the second step, system initialization, the Kernel has defined a simple initialization protocol (shown in Figure 15-2). This protocol requires that one processor, called the Master, be in charge of the initialization process. All other processors in the network are subordinate to this processor during the Kernel's initialization process. The Master is responsible for:

- Ensuring the consistency of the NCT among all the subordinate processors.
- Issuing the "Go" message to all the subordinate processors.

Some key points to note about this protocol are:

- The Master processor is a single point-of-failure in the system.
- The Master assumes it has the correct and complete version of the Network Configuration Table.
- If any of the following problems occurs at initialization, then the network may fail to become operational:

    1. No Master processor declares itself.
    2. The Master processor fails to initialize successfully.
    3. More than one Master processor declares its presence.
    4. The Network Configuration Tables are found to be inconsistent.

These points can be addressed by application-specific fault tolerant techniques (redundant hardware, voting schemes, etc.), which are in the domain of the application, not the Kernel. This is discussed further in the *Kernel User's Manual.*

15.1. Primitives

15.1.1. Initialize Master processor

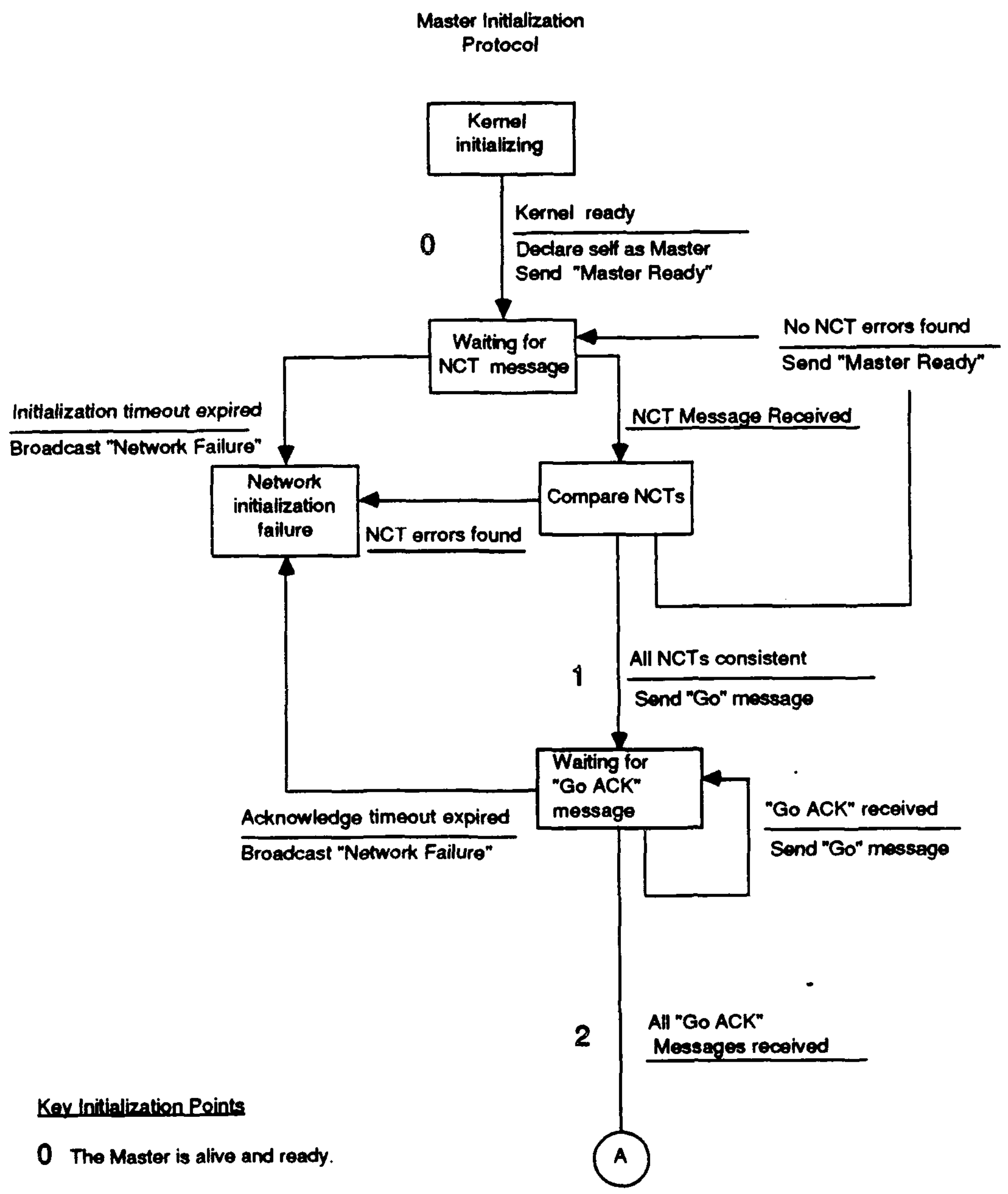
This primitive identifies the invoker as the processor that is going to control network initialization. It causes the Kernel to initiate the Master initialization protocol shown in part one of Figure ı5-2. It requires a timeout that is used to control how long the Master processor will wait for any one subordinate to reply to any initialization protocol message. The expiration of this timeout informs the Master processor that network-wide initialization has failed; it is the responsibility of the Main Unit to relay this information to the user.

The initialization protocol shown in Figure 15-2 consists of two phases:

1. Network phase: The Master processor interrogates each subordinate to determine its view of the network (embodied in the NCT).
2. Commence processing phase: The Master processor tells each subordinate to start normal processing.

This primitive synchronizes the clocks on each of the subordinate processors with the Master processor's clock.

This primitive can give rise to the following status codes:



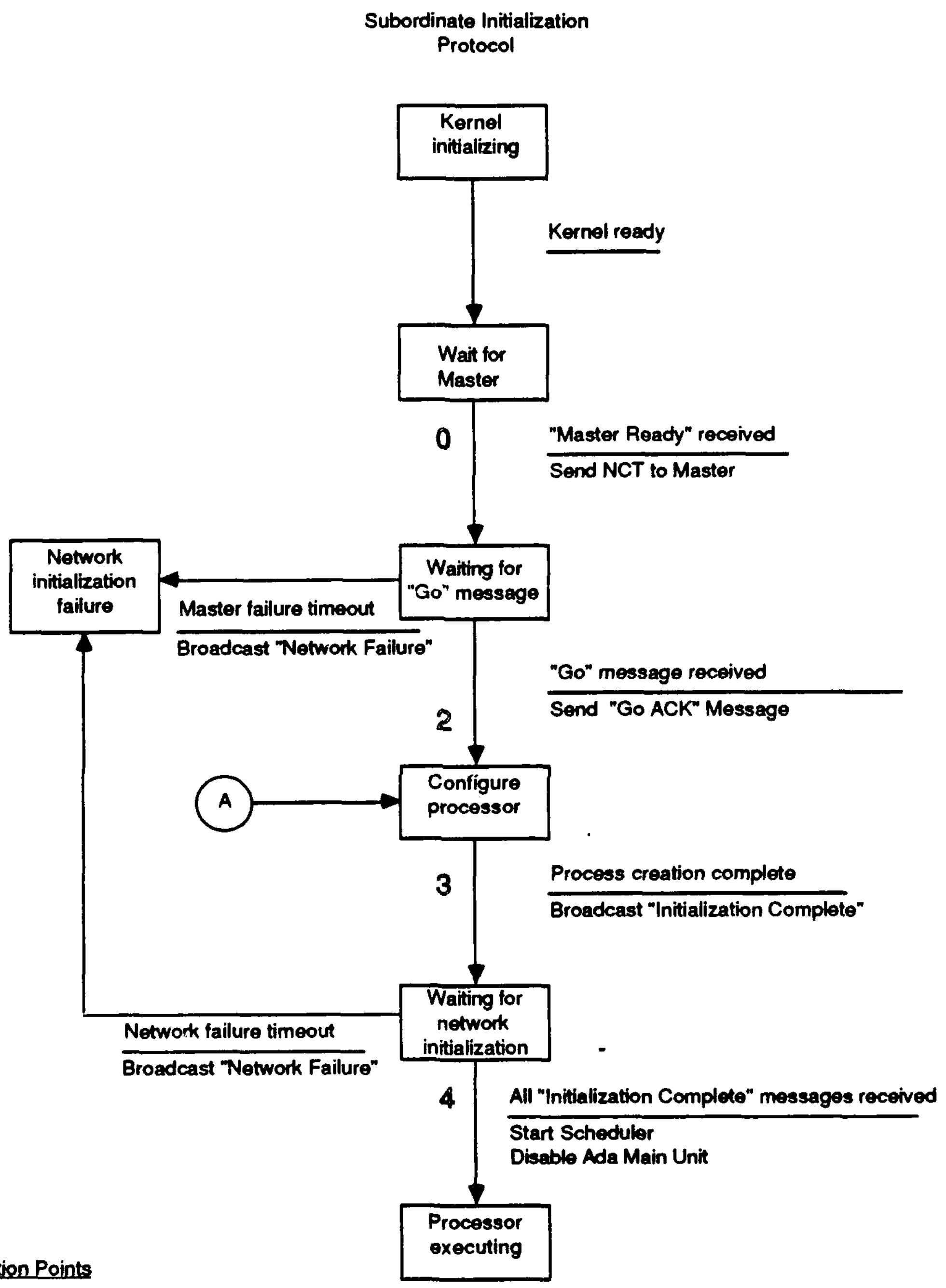
Key Initialization Points

0 The Master is alive and ready.

1 The Master knows that the physical topology of the network is consistent across processors.

2 All processors know that the physical topology of the network is consistent.

Figure 15-2: Network Initialization Protocol (Part 1 of 2)



<u>Key Initialization Points</u>

3	Processor configuration is complete; waiting for remaining processors to complete.
4	All processors know that the logical topology of the network is consistent. The application is ready to execute.

Figure 15-2: Network Initialization Protocol (Part 2 of 2)

- OK
- Calling unit not Main Unit
- Configuration tables inconsistent
- Master initialization timeout expired
- Network failure
- Processor failed to ACK go message
- Processor failed to transmit NCT

15.1.2. Initialize subordinate processor

This primitive identifies the invoker as a subordinate (i.e., non-Master) processor and begins execution of the *subordinate initialization protocol* shown in part two of Figure 15-2. This primitive has an optional timeout parameter, which is used to set a bound on how long the subordinate will wait for the Master processor to initialize. If this timeout expires, an alternative Master processor may be designated.

This primitive can give rise to the following status codes:

1. OK
2. Calling unit not Main Unit
3. Network failure
4. Subordinate initialization timeout expired

15.1.3. Start subordinate processors

Deleted - 1 July 1988.[25]

15.1.4. Create network configuration

This primitive creates the Network Configuration Table shown in Figure 15-1. This creation is a static, compile-time operation performed by the Kernel user. A complete copy of the NCT must exist on every Kernel processor in the network.

This primitive always succeeds.

15.2. Blocking Primitives

This section lists which of the primitives described above may block the invoking process and the conditions under which they will block:

1. Initialize Master processor: Always blocks until one of the following conditions occurs:
    a. All processors in the network required for initialization have transmitted

[25]Subsumed by Initialize Master Processor (15.1.1).

their NCTs to the Master and have acknowledged the "Go" message, or

b. The initialization timeout expires.

2. Initialize subordinate processor: Always blocks until one of the following conditions occurs:

   a. The Master has requested the processor's NCT and the subordinate has acknowledged the "Go" message, or

   b. The initialization timeout expires.

15.3. Status Codes

1. Calling unit not Main Unit: Informs the invoker that this primitive may be invoked only by the Main Unit on a processor (and hence only during initialization).
2. Configuration tables inconsistent: A discrepancy was found between the Network Configuration Table of a subordinate processor and the Network Configuration Table of the Master processor.
3. OK: Normal, successful completion.
4. Processor failed to ACK go message: Indicates that a processor has failed at system startup time.
5. Processor failed to transmit NCT: Indicates that a processor has failed at system startup time.
6. Master initialization timeout expired: Informs the invoker that the Master failed to completely initialize the network within its alloted time.
7. Subordinate initialization timeout expired: Informs the invoker that the subordinate failed to receive the correct sequence of messages from the Master processor within its alloted time.
8. Network failure: Informs the invoker of a failure in the network communications medium where the reason for the failure cannot be determined by the primitive.

16. Process Management

This chapter outlines the primitives provided to the Ada application for creation and termination of Kernel processes. The model used for Kernel process names is this: to communicate with a process, one must know the name of the process either at compile time or as constructed at runtime. The name of a process comes in two forms:

1. The *logical name* given to the process by the developer, encoded in Ada as a character string, and
2. The internal name given to the process at runtime by the Kernel.

Hereafter, the internal name of a process is called the *process ID* (PID) or *process identifier*; the term *process name* refers to the logical name of the process. However, knowing the name of a process does not guarantee the availability of the process at runtime. This is one class of faults that the Kernel is able to detect and report.

16.1. Primitives

16.1.1. Declare process

This primitive declares all Kernel processes that will execute locally, all remote Kernel processes with which communication may occur, and any remote, non-Kernel devices with which communication will occur. This primitive may be invoked only by the Main Unit.

This primitive can give rise to the following status codes:

- OK
- Calling unit not Main Unit
- Insufficient space
- Process already exists
- Unknown non-Kernel device

16.1.2. Create process

This primitive creates an independent thread of control. It may be invoked only by the Main Unit of a processor to create the Kernel processes that will share that processor. For each process, the application provides the following information to the Kernel:

- Process attributes: stack size, and code address
- Scheduling attributes: priority and preemption[26]
- Communication attributes: maximum message size, message queue size, and message queue overflow handling

Once the Kernel has all the process-related information, it constructs the execution environment, shown in Figure 16-1, around the process. This environment consists of:

[26] A process will always be able to modify its own scheduling information at a later time.

- *Process stack*, containing:
    - Stack plug
    - Dummy call frame
    - Local process variables
- *Process information record*, containing:
    - Message queue
    - Schedule attributes
    - Process code
    - Context save area

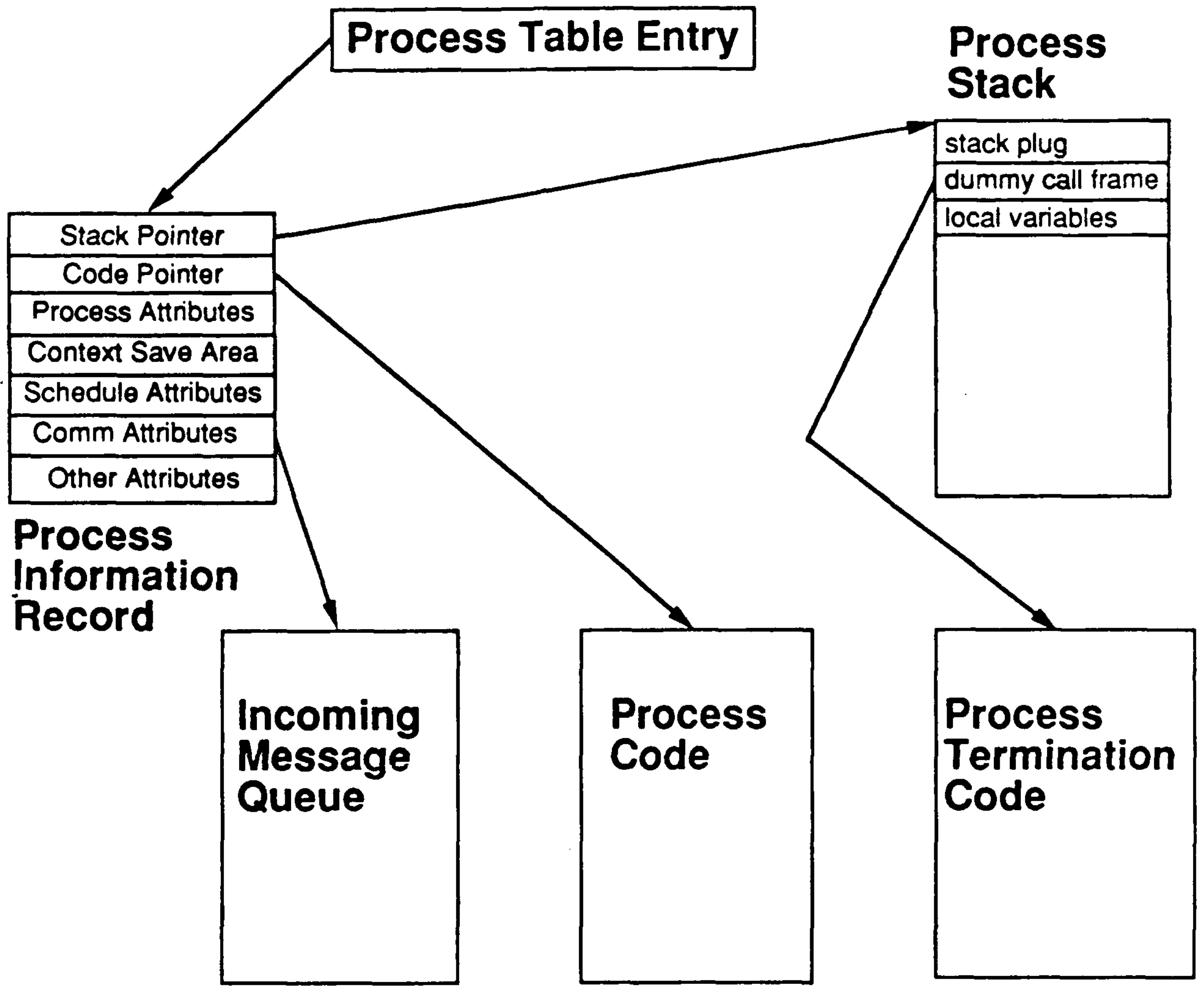


Figure 16-1: Process Execution Environment

This primitive can give rise to the following status codes:

- OK
- Calling unit not Main Unit
- Illegal process address
- Illegal process identifier
- Insufficient space
- No Kernel process on non-Kernel device
- Process already created

16.1.3. Initialization complete

This primitive asserts that the declaration and creation of all processes on this processor is now complete. It is invoked by the Main Unit of that processor after all processes have been declared and/or created. This primitive effectively tells the Kernel "I'm alive and ready to roll!" — the application code is ready to run — and the Kernel relays this information to all the other Kernels in the network. This primitive takes an optional timeout parameter to detect processor failure after network initialization.

This primitive can give rise to the following status codes:

- OK
- Calling unit not main unit
- Final sync initialization timeout expired
- Network failure
- Process initialization failure
- Process maximum exceeded

16.1.4. Allocate device

This primitive assigns a specific process to be the recipient of all messages originating from a specific non-Kernel device although any process can send a message to a non-Kernel device.[27]

This primitive can give rise to the following status codes:

- OK
- No such device exists
- Replacing previous allocation

[27]Of course, it must be properly formatted for the device by the application.

16.1.5. Die

This primitive terminates the calling process (self-termination only) and may be invoked by any process at any time. Since processes have no dependents, each one terminates individually. Messages pending when a process terminates are discarded, as are messages that arrive after a process has terminated. All other resources held by the process are also released. Note that only Kernel processes may be terminated by this primitive. Terminating a non-Kernel process is the user s responsibility.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call

16.1.6. Kill

This primitive terminates the specified process and may be invoked by any process at any time. This can be applied to any named process, including the calling process. This is an "abnormal" termination, and causes an immediate action by the Kernel. If the terminated process is remote, the actual processing occurs after return from this primitive on the terminated process's host node. Messages pending when a process is terminated are discarded, as are messages that arrive after a process has terminated. All other resources held by the process are also released.

This primitive always succeeds.

16.1.7. Who am I

This primitive allows a process to obtain its own process identifier and may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call

16.1.8. Name of

This primitive allows a process to obtain the logical name of a process for which it has a process ID. It may be invoked by any process at any time.

This primitive always succeeds.

16.2. Blocking Primitives

This section lists which of the primitives described above may block the invoking process and the conditions under which they will block:

1. Initialization complete: Always blocks until one of the following conditions occurs:
   a. All the needed initialization complete messages are received, or
   b. The initialization timeout expires.

16.3. Status Codes

1. Calling unit not Main Unit.
2. Final sync initialization timeout expired.
3. Illegal context for call: The requested operation cannot be performed by an interrupt handler.
4. Illegal process address: The code address of the process is not a valid Ada address.
5. Illegal process identifier: A nonexistent process identifier was used.
6. Insufficient space: Insufficient memory exists for the creation of another process.
7. Network failure.
8. No Kernel process on non-Kernel device.
9. No such device exists: The requested non-Kernel device is not known at the requesting site.
10. OK.
11. Process already created.
12. Process already exists.
13. Process initialization failure.
14. Process maximum exceeded: Declaration exceeded maximum number of processes allowed.
15. Replacing previous allocation: The current process allocated to the device is replaced by the invoking process.
16. Unknown non-Kernel device: No such device identified in the NCT.

17. Semaphore Management

The Kernel provides the traditional Boolean ("Dykstra") semaphore facility, slightly revised to be consistent with the overall philosophy of the Kernel primitives.

A semaphore is an abstract data type. Objects of this type may be declared anywhere, but since semaphores are used to build process synchronization systems, they are clearly best declared in the Main Unit of a processor. A semaphore is visible only on the processor on which it is declared, and therefore can be used only by processes local to that processor, for example for application-level control of shared memory.

At any time, a semaphore is in one of two states:

- FREE: The semaphore is free, or
- CLAIMED(N): The semaphore is claimed, and N processes are awaiting its release. These processes are blocked on a FIFO queue associated with the semaphore.

17.1. Primitives

17.1.1. Declare semaphore

A semaphore is declared by a normal Ada declaration. Its initial state is FREE.

This primitive always succeeds.

17.1.2. Claim semaphore

This primitive attempts to claim the semaphore. If the semaphore was free, the primitive succeeds, the semaphore state changes to CLAIMED(0), and the invoking process continues.

If the semaphore was CLAIMED(N), then the invoking process blocks. The state changes to CLAIMED(N+1), and the process is appended to the semaphore queue. The call can optionally specify a timeout and a resumption priority.

This primitive can give rise to the following status codes:

- OK
- Claim timed out
- Illegal context for call

17.1.3. Release semaphore

This primitive attempts to release a semaphore previously claimed. The state necessarily must be CLAIMED(N). If N=0, no other process is waiting, and the semaphore becomes FREE. Otherwise, the state becomes CLAIMED(N-1), and the process at the head of the semaphore queue is given the semaphore and becomes suspended.

Note that a release cannot block, but may cause the invoking process to be preempted if the process at the head of the queue has a higher priority.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call
- Not my semaphore

17.2. Blocking Primitives

This section lists which of the primitives described above may block the invoking process and the conditions under which they will block:

1. Claim: Blocks only when the state of the requested semaphore is CLAIMED(N). It will unblock when one of the following conditions occurs:
    a. The semaphore is released and the invoking process is at the head of the semaphore wait queue, or
    b. The claim timeout expires.

17.3. Status Codes

1. OK.
2. Claim timed out: Operation unblocked due to timeout expiration.
3. Illegal context for call.
4. Not my semaphore: Attempt to release a semaphore that has not previously been claimed by the invoker.

18. Schedule Management

This chapter outlines the basic scheduling mechanisms to be provided by the Kernel. The scheduling paradigm used by the Kernel is a simple, prioritized, event-driven model that permits the construction of preemptive, cyclic, and non-cyclic processes. To achieve this, there are four types of events in this model:

1. Receipt of a message (synchronous event).
2. Receipt of a message acknowledgment (asynchronous event).
3. Expiration of a primitive timeout (asynchronous event).
4. Expiration of an alarm (asynchronous event).

The scheduling primitives are discussed below, and the alarm primitives are discussed in Chapter 22. This paradigm allows the user to create:

- A non-cyclic process that executes until preempted by a higher-priority process.
- A set of non-cyclic processes that execute in a round-robin, time sliced manner.
- An event-driven process that blocks when trying to receive a message. It is resumed from the point of suspension, when it is able to proceed and when the priority admits.
- An event-driven process that blocks itself for a specified period of time (or equivalently, until a specific time) and is resumed at a specific priority (this allows a "hard" delay to be implemented).
- A cyclic process that continuously executes a body of code (and that can detect frame overrun).

To support these paradigms, the following set of scheduling attributes is defined:

- Priority:
  - Every process has a priority.
  - Priorities are relative within one processor.
  - Priorities are incommensurable across processors.
  - A process may change its priority dynamically.
  - Priorities are strict and pre-emptive; *higher-priority processes always shut out lower-priority processes.*
  - Blocking primitives allow the caller to specify a resumption priority, which may be different from the priority at the point of invocation. The resumption priority is the priority of the process when it unblocks.

- Timeslice:
  - The maximum length of time a process may run before another process of the same priority is allowed to run.
  - A property of a set of processes on the same processor and all of the same priority.
  - *Time slicing cannot override priority;* it applies only among processes of equal priority.

- Any process may enable or disable time slicing for the entire processor.
- Any process may set the timeslice quantum.
- A process may allow (or disallow) itself to be sliced by setting its preemption status (if preemptable, the process may be time sliced; if not preemptable, the process may not be time sliced by another process of the same priority).

Given this scheduling regime, a process is always in one of four states:

1. *Running*: A running process is executing on its processor, and it continues to run until something happens. If interrupts are enabled, they occur transparently unless they cause a change of process state. A running process ceases to run when it: dies, invokes a blocking Kernel primitive, is time sliced, is killed by another process, or is preempted by a higher-priority process. The first three are voluntary actions on the part of the process, while the last two are actions performed by the Kernel.
2. *Dead*: A dead process is unable to run again. A process dies in one of five ways: by completing execution, by raising an unhandled exception, by causing an unrecoverable error, by killing itself, or by being killed by another process. Processes are not expected to die, and any subsequent attempts to interact with a dead process result in errors.
3. *Blocked*: A blocked process is unable to run. A process may only become blocked as a result of its own actions. These blocking actions are waiting for: the arrival of a message, the arrival of a message acknowledgment, the completion of a clock synchronization, a specific duration, a specific time, or the availability of a semaphore. A process becomes unblocked when the blocking event occurs (at which time the process transitions to the suspended state). An unblocked process does not immediately resume execution; it resumes execution only when the Scheduler so decides. However, the process can affect this decision by specifying a resumption priority in the primitive invocation.
4. *Suspended*: A suspended process is able to run, but cannot run because a process of higher or equal priority is running. A process may be resumed when the running process blocks, lowers its own priority, or is time sliced.

These states and the transitions between them are shown in Figure 18-1.

For instance, a *running* process becomes blocked by trying to receive when no message is pending. It becomes unblocked (but *suspended*) when the message arrives. It becomes *running* when its priority permits. A *running* process can also call wait, to *block* itself at any time. The waiting process becomes *suspended* and thus ready to run when its delay expires. Further, a running process may be preempted, that is, forcibly *suspended* against its will, to allow a higher-priority process to *run* or to be time sliced.

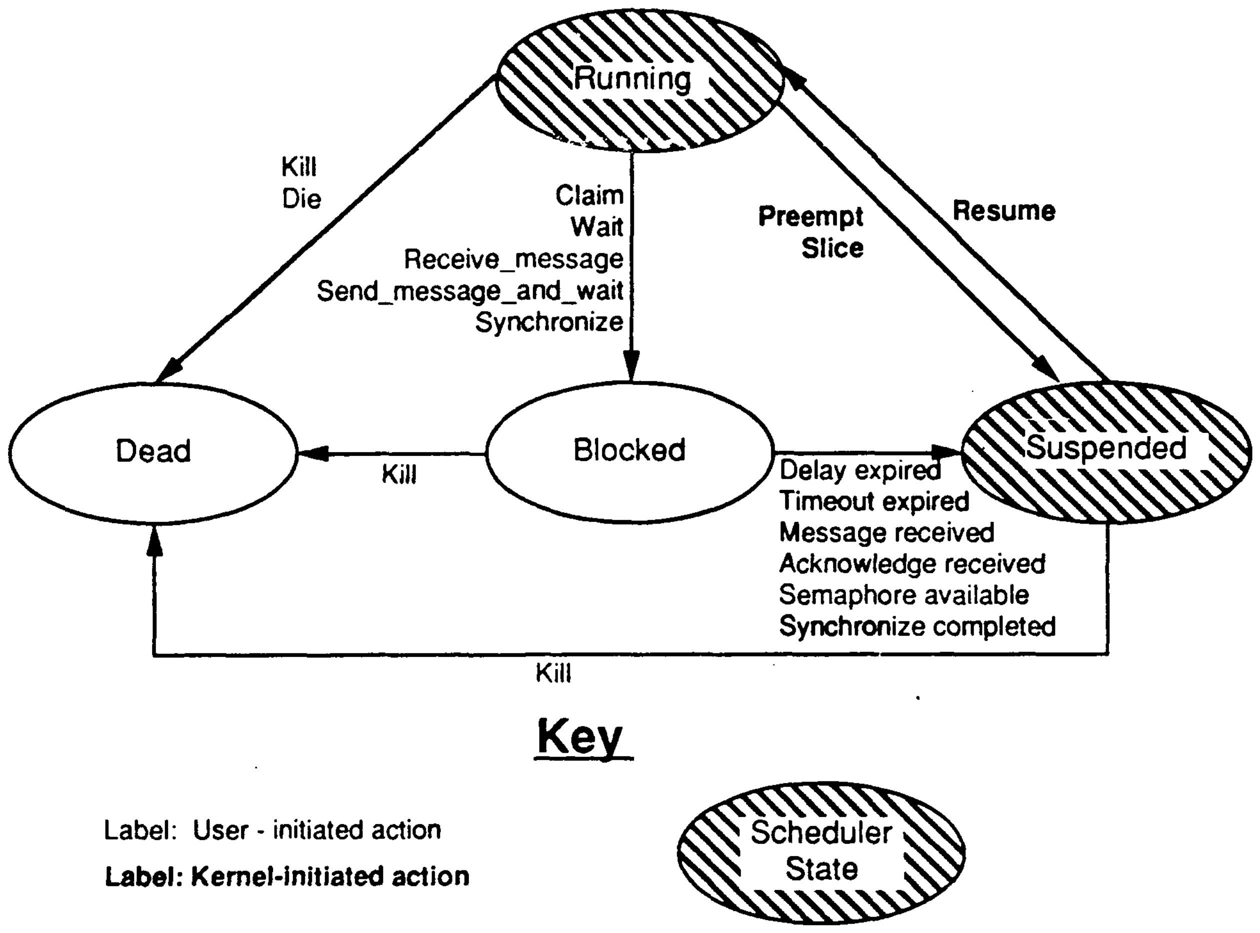


Figure 18-1: Process States

18.1. Primitives

18.1.1. Set process preemption

This primitive changes the preemption state of the calling process. A process may set only its own preemption state. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call

18.1.2. Get process preemption

This primitive queries the current value of the preemption state of the calling process. A process may query only its own preemption state. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call

18.1.3. Set process priority

This primitive changes the priority of the calling process. A Kernel process may set only its own priority. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call

18.1.4. Get process priority

This primitive queries the current value of the priority of the calling process. A process may query only its own priority. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call

18.1.5. Wait

This primitive allows the invoker to suspend its own execution:

- Until a specified time, or
- For a specified duration.

An optional resumption priority may also be specified. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call

18.1.6. Set timeslice

This primitive sets the timeslice quantum for the processor.

This primitive can give rise to the following status codes:

- OK
- Illegal timeslice

18.1.7. Enable timeslicing

This primitive enables the Kernel to perform round-robin, timeslice scheduling among processes of equal priority.

This primitive always succeeds.

18.1.8. Disable timeslicing

This primitive disables round-robin, timeslice scheduling. After execution of this primitive, scheduling is priority-based preemption.

This primitive always succeeds.

18.2. Blocking Primitives

This section lists which of the primitives described above may block the invoking process and the conditions under which they will block:

1. Wait: Always blocks until the delay (i.e., the specified absolute or elapsed time) expires.

18.3. Status Codes

1. OK.
2. Illegal context for call.
3. Illegal timeslice.

19. Communication Management

This chapter outlines the primitives provided for communication between Kernel processes (see Chapter 15 for details about how the communication model ties into the network model). The communication model is based on the following premises:

- All communication is point-to-point.
- A sender must specify the recipient.
- A recipient gets all messages and is told the sender of each.
- A recipient cannot ask to receive only from specific senders.
- Messages do not have priorities.

The purpose of a message is to convey information between processes. To the Kernel, a message is just a sequence of uninterpreted bits. The Kernel provides the untyped primitives; the users may build above them whatever application-specific functionality is needed. Note that any two (or more) communicating processes are free to define a common package containing their interface message types. Communication between processes on a single processor is optimized.

Figure 19-1 illustrates this communication model. In this figure, process Merlin on *Processor a* sends a message to process Vivian on *Processor b*. This is accomplished by Merlin's informing the Kernel of the message content and the logical destination of the message (i.e., Vivian). The Kernel on *Processor a* takes this message, formats the datagram to hold the message, and transmits the datagram over the network to *Processor b*, where it knows Vivian resides. When the message arrives at *Processor b*, the Kernel there rebuilds the message from the datagram and queues it for Vivian until Vivian requests the next message. If Merlin had wanted acknowledgment of message receipt by Vivian, the Kernel on *Processor b* would have formatted an acknowledgment datagram and sent it back to *Processor a* after Vivian had asked for (and received) the message.

19.1. Primitives

19.1.1. Send message

This primitive is used to send a message from one process to another, without waiting for acknowledgment of message receipt. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Receiver dead
- Receiver never existed

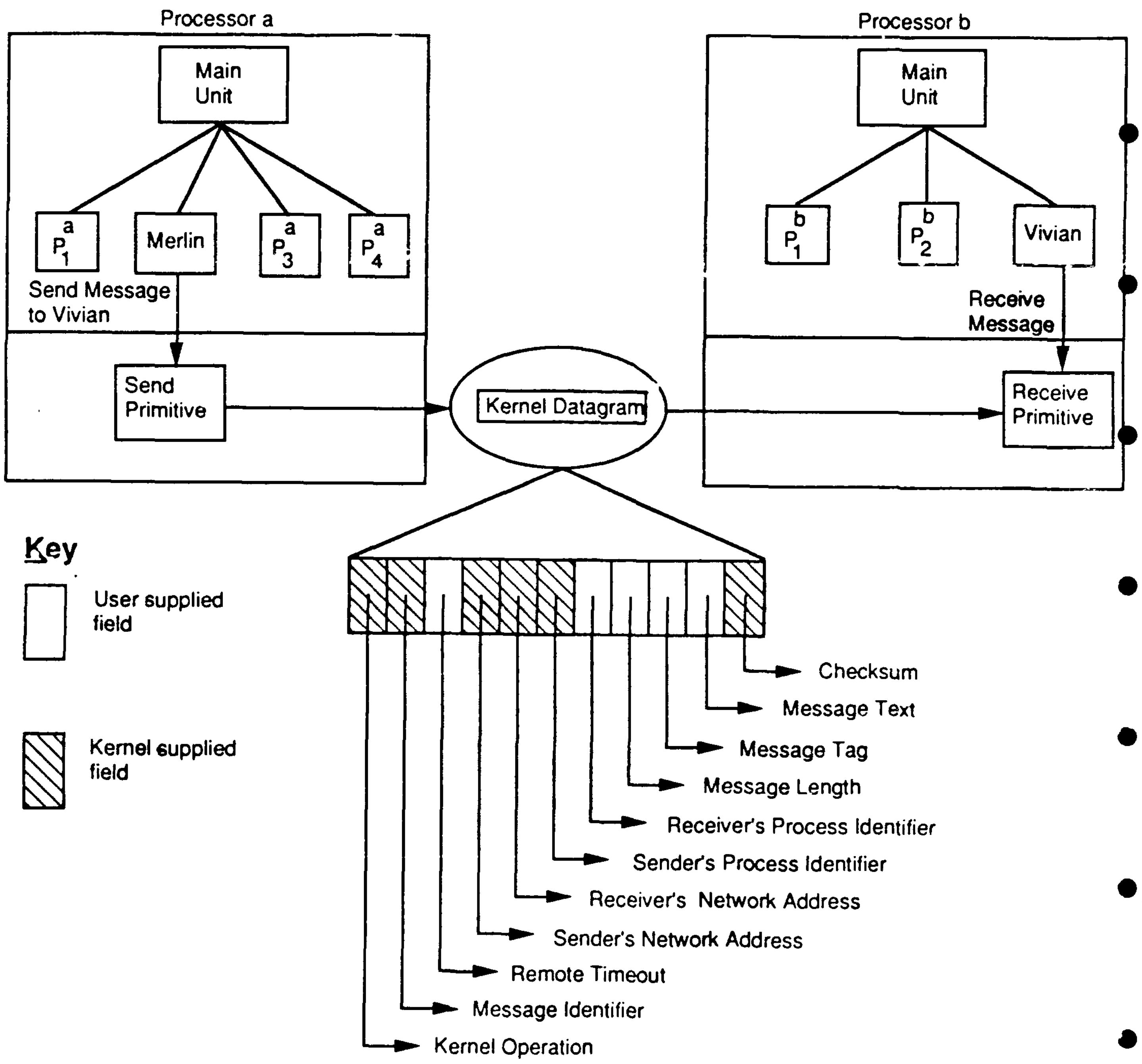


Figure 19-1: Datagram Network Model

19.1.2. Send message and wait

This primitive is used to send a message from one process to another; the sender blocks while waiting for acknowledgment of message receipt by the receiving process. This primitive may be invoked by any process at any time. However, it may not be invoked by an interrupt handler. An optional timeout may be specified. If a timeout is specified, the Kernel performs a remote timeout; that is, the timeout is bundled with the message and executed by the Kernel of the receiving process. If the timeout expires, the message is purged from the receiver's message queue and the invoking process is notified. The Kernel rejects all calls where the recipient is the sender.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call
- Message not received
- Message timed out
- Network failure
- Receiver dead
- Receiver is sender
- Receiver never existed

19.1.3. Receive message

This primitive is used to receive a message from another process. This primitive may be invoked by any process at any time. The Kernel automatically performs any required acknowledgments. An optional timeout may be specified. If a timeout is specified, the Kernel performs a local timeout. If the timeout expires, the invoking process is notified. If this primitive is invoked with a zero-duration timeout, it does not block, but returns immediately with a status code should no message be available. This primitive may not be invoked by an interrupt handler.

This primitive can give rise to the following status codes:

- OK
- Buffer too small for message
- Illegal context for call
- Message timed out
- No message available

19.2. Blocking Primitives

This section lists which of the primitives described above may block the invoking process and the conditions under which they will block:

1. Send message and wait: Always blocks until one of the following conditions occurs:

    a. The receiving process has requested and received the message, or
    b. The message timeout expires.

2. Receive message: Blocks only if there is no message currently available for the process and a positive timeout has been specified. If no message is available, then it blocks until one of the following conditions occurs:

a. A message has arrived for the process, or

b. The timeout expires.

19.3. Status Codes

1. OK.
2. Buffer too small for message.
3. Illegal context for call.
4. Message timed out: Timeout expired without operation completing.
5. Message not received: The message was discarded from the receiver's message queue.
6. Network failure.
7. No message available: Occurs in situations where a zero timeout is specified.
8. Receiver dead: Destination process has terminated or been aborted.
9. Receiver is sender: A process cannot perform an acknowledged send with itself.
10. Receiver never existed.

20. Interrupt Management

This section outlines the interrupt control primitives provided by the Kernel. There are two parts to the Kernel's view of interrupts: interrupts themselves and interrupt handlers. The interrupt model used by the Kernel is based on the following premises:

- There are devices that can interrupt the processor.
- There are three classes of interrupts:
    1. Those reserved by the Ada runtime environment (divide-by-zero, floating-point overflow, etc.).
    2. Those reserved by the Kernel (such as the clock interrupt).
    3. Those available to the user (everything not in 1 and 2 above).

  All the primitives described below apply only to the third class of interrupts.
- The device interrupt may be either enabled or disabled. If the interrupt is disabled, the device cannot interrupt, regardless of how badly it might want to.
- The Kernel does not queue interrupts nor does it hide hardware-level interrupt properties, such as queueing of interrupts, interrupt priorities, or non-maskable interrupts.
- Interrupts are events local to a processor and cannot be directly handled or bound by processes running on a different processor.

The model used for interrupt handlers is:

- An interrupt handler is an Ada procedure with no parameters or some other code following the same procedure-call conventions as an Ada procedure.
- Interrupt handler code can access procedure local or processor global memory.
- Interrupt handler code has access to all the Kernel primitives; the only restriction is that a handler is not allowed to block its own execution.
- If an interrupt is enabled and a handler is bound, then the occurrence of the interrupt transfers control to the bound handler, which is code the user has supplied.

The primitives that implement these models are discussed below.

20.1. Primitives

20.1.1. Enable

This primitive allows the specified interrupt to occur. No interrupt can be enabled via the Kernel unless the Kernel has bound a handler for that interrupt. This implies that there may be handlers bound outside the knowledge of the Kernel. This is legitimate, since the Kernel is only responsible for those handlers that it binds. This primitive may be invoked by any process at any time (including the Main Unit).

This primitive can give rise to the following status codes:

- OK
- Illegal interrupt
- No interrupt handler bound
- Reserved interrupt

20.1.2. Disable

This primitive prohibits the specified interrupt by ignoring its occurence. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal interrupt
- Reserved interrupt

20.1.3. Enabled

This primitive queries the status of the specified interrupt (i.e., whether it is enabled or disabled). This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal interrupt
- Reserved interrupt

20.1.4. Simulate interrupt

This primitive simulates the occurrence of a specified interrupt in software. An interrupt handler must be bound to the specified interrupt for this primitive to have an effect. This primitive may be invoked by any process at any time. This primitive returns only after the interrupt handler has completed its processing.

This primitive can give rise to the following status codes:

- OK
- Illegal interrupt
- No interrupt handler bound
- Reserved interrupt

20.1.5. Bind interrupt handler

This primitive associates the specified interrupt with the Ada procedure[28] identified as the interrupt handler. This primitive may be invoked by any process at any time (including the Main Unit). Any attempt to bind a different interrupt handler to a bound interrupt results in the old handler being replaced. Invocation of this primitive causes the Kernel to construct an execution environment for the handler, as shown in Figure 20-1. The environment consists of:

- Kernel encapsulation
- Interrupt handler code
- *Interrupt table entry*, containing:
    - Handler code
    - Handler status
    - Interrupt status
    - Interrupt vector address

This primitive can give rise to the following status codes:

- OK
- Illegal interrupt
- Illegal interrupt handler address
- Replacing previous interrupt handler
- Reserved interrupt

20.2. Blocking Primitives

None.

20.3. Status Codes

1. OK.
2. Replacing previous interrupt handler.
3. No interrupt handler bound: An interrupt cannot be enabled for which no handler has been defined (i.e., bound).
4. Illegal interrupt.
5. Reserved interrupt.
6. Illegal interrupt handler address: The address specified for the interrupt handler code is not a legal Ada address.

[28]Or an equivalent procedure in another language.

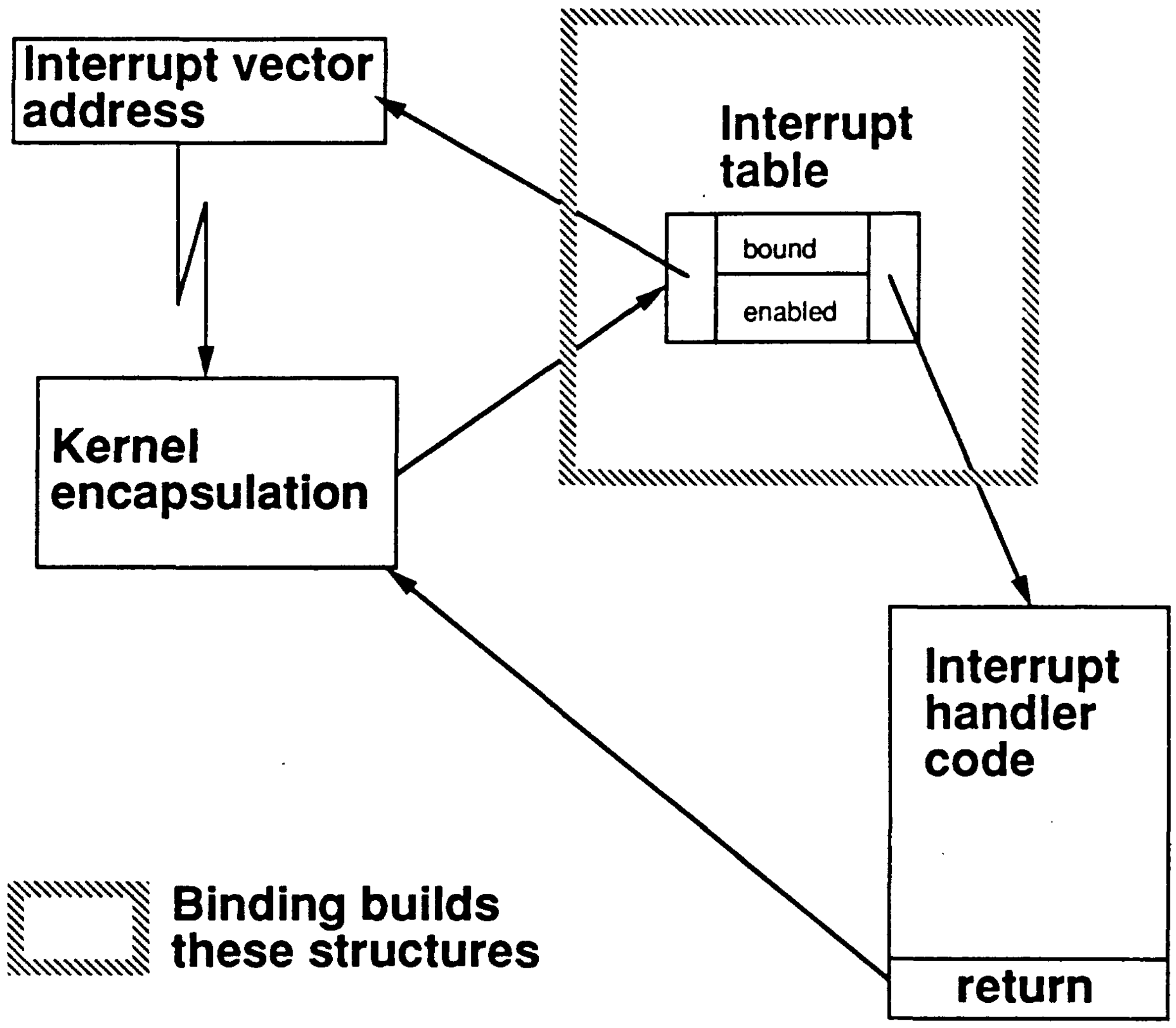


Figure 20-1: Interrupt Handler Execution Environment

21. Time Management

The concept of time permeates the entire Kernel. Many of the Kernel concepts and primitives rely on time, specifically:

- Network management uses time for initial clock synchronization and for timeout parameters in the primitives: *initialize master processor* and *initialize subordinate processor.*
- Process management requires time for a timeout parameter, in the *initialization complete* primitive.
- Schedule management uses time for round-robin, timeslice scheduling and for delays via the *wait* primitive.
- Communication management requires time for timeout operations in *receive message* and *send message and wait* primitives.
- Semaphore management requires time for a timeout parameter, in the *claim* primitive.
- Time management requires time for a timeout parameter, in the *synchronize* primitive.
- Alarm management uses time for setting alarms via the *set alarm* primitive.

To support these primitives, the Kernel contains facilities for time management, both for its own use and to make available to the application code. In all cases, two forms of delay are available to the application:

- Elapsed time: This computes the delay as elapsed time from the moment the primitive is called. This delay is similar to a value of the Ada type *duration.*
- Epoch time: This delays until a specified time of resumption. This delay is similar to a value of the Ada type *time.*

The rationale for the two kinds of time is that they express fundamentally different concepts. For example, if a certain action should be performed daily at midnight, it is not correct to perform the action "every 24 hours," since successive midnights are not always 24 hours apart. Similarly, if an action should be performed every 5 minutes, it is not correct to schedule three such actions for 0155, 0200, and 0205, since 65 minutes might elapse between the second and third (i.e., the clock might have been reset).

The application programmer must be able to choose the type of time representation needed. Resetting the system time affects the two types of timeouts differently.

In the current design, the assumption is made that it is feasible for all the target processors to use a common time base and to record the passage of time at the same uniform rate. It must be recognized that there are some real-time applications for which this assumption is unrealistic, since the processors will be distributed across several different inertial frames of reference, but it will serve for this prototype.

It is necessary therefore to describe the local representation of time and the clock synchronization mechanisms.

Representation of Time

At any moment, on any processor, the current time is given by a combination of three values:

1. Elapsed. The elapsed time is the number of ticks since the end of the application initialization process.
2. Epoch. The epoch is a value representing the time-of-day—the moment at which the processors began to compute elapsed time.
3. Base. The base is the calendar date corresponding to an epoch of zero, i.e., the base of the representation of time. Julian Day 1 (started at 1200 on 1 January 4713BCE) has been chosen.[29]

The representation chosen for both epoch time and elapsed time is fine enough to allow accurate measurement and large enough to allow code to run for a very long time. Thus the current time of day = Base + Epoch + Elapsed.

Time is set initially on the Master processor in the network by the application. This is done either by hand, during operator dialogue, or by reading a continuously running hardware device. This time is then communicated to the subordinate processors during system initialization. The processors may then synchronize system time by having this processor use the synchronize primitive discussed below. This gives the application complete control over when to synchronize system time. Once the clocks are synchronized, the Kernel does not attempt to maintain the synchronization. The processors resynchronize only as a result of deliberate action by application code.

Three forms of time adjustment are supported:

1. The elapsed time for any processor can be changed by an explicit command. This is to be used when one processor's time computation has gone awry. It has the effect of changing pending timeouts of either kind, since increasing the number of elapsed ticks makes the machine think both that it has been running longer and that it is later in the day.
2. The epoch time of any processor may be changed. This is to be used if it is discovered that the original time setting was wrong. It has the effect of changing any pending epoch timeouts, since increasing the epoch makes the machine think it is later in the day, but does not change how long it thinks it has been running.
3. The Kernel provides a primitive that explicitly synchronizes all the clocks in the network, as defined in Section 21.1.6.

[29]Thus, 1 January 1988 is JD 2447162.

21.1. Primitives

21.1.1. Package calendar

The Ada package Calendar is supported. The existing vendor-supplied package is available. The Kernel does not use package Calendar internally, so applications are not required to include it in the load modules.

21.1.2. Time constants

The Kernel defines the constants:

- TICK: smallest resolvable interval of time.
- SLICE: smallest resolvable scheduling interval of time.

The Kernel maintains time internally accurate to the TICK. The definition of TICK is local to a processor, and no relationship between this definition is assumed or required across the processors in the network. However, the internal represenation of time is consistent across processors (see requirement 12.1.25).

21.1.3. Adjust elapsed time

This primitive allows the application to increment or decrement the current local elapsed time by a specified number of clock ticks. This primitive may be invoked by any process at any time.

This primitive can give rise to one of the following status codes:

1. OK.
2. Change reseults in negative elapsed time

21.1.4. Adjust epoch time

This primitive allows the application to reset the epoch time of the local processor clock. This primitive may be invoked by any process at any time.

This primitive can give rise to one of the following status codes:

1. OK.
2. Change results in negative epoch time
3. OK, but requested time already passed.

21.1.5. Read clock

This primitive reads the local processor clock and returns the current time-of-day as an epoch time.

This primitive always succeeds.

21.1.6. Synchronize

This primitive forces all local processor clocks on Kernel devices to synchronize time with the local clock on the invoking processor. This primitive takes an optional timeout parameter. This primitive may be invoked by any process at any time.

The post-conditions of this primitive are:

1. If it completes successfully, all clocks are synchronized.
2. If it terminates with an error, the exact state of network time is not known.

This primitive can give rise to one of the following status codes:

1. OK
2. Network failure
3. Synchronization in progress
4. Synchronization timeout

21.2. Blocking Primitives

This section lists which of the primitives described above may block the invoking process and the conditions under which they will block:

1. Synchronize: This primitive always blocks the invoker until:
    a. All Kernel clocks are synchronized,
    b. An error condition is detected, or
    c. The timeout expires.

21.3. Status Codes

1. OK.
2. Change results in negative elapsed time.
3. Change results in negative epoch time.
4. OK, but requested time already passed.
5. Network failure: A processor in the network fails to respond appropriately to the synchronization protocol.
6. Synchronization in progress: A processor attempts to synchronize time while a previous invocation of the synchronization primitive is in progress.
7. Synchronization timeout: The synchronization protocol does not complete before the timeout expires.

22. Alarm Management

This chapter outlines the primitives for alarm management. Alarms are:

- Enforced changes in process state.
- Caused by the expiration of a timeout.
- Asynchronous events that are allocated on a per-process basis (each process may have no more than one alarm).

Processes view alarms as a possible change in priority with an enforced transfer of control to an exception handler. Alarms are requested to expire at some specified time in the future. When an alarm expires, the Kernel raises the **alarm_expired** exception,[30] which the process is expected to handle as appropriate. Note that if a zero or negative duration or an absolute time in the past is specified, the alarm expires immediately.

22.1. Primitives

22.1.1. Set alarm

This primitive defines an alarm that will interrupt the process if it expires. An optional resumption priority may be specified. If the alarm expires, the Kernel raises the **alarm_expired** exception (when the process is *running*), and control passes to the exception handler of the process. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call
- Resetting existing alarm

22.1.2. Cancel alarm

This primitive turns off an alarm that was set but has not yet expired. This primitive may be invoked by any process at any time.

This primitive can give rise to the following status codes:

- OK
- Illegal context for call
- No alarm set

[30] When the process's priority permits it to be scheduled.

22.2. Blocking Primitives

None.

22.3. Errors

1. OK.
2. Illegal context for call.
3. Resetting existing alarm.
4. No alarm set.

23. Tool Interface

The Kernel is a utility intended to support the building of distributed Ada applications. As such, it is important the the Kernel be able to work in harmony with user-developed support tools. To provide that support, the Kernel must provide a window into its internal workings. It is envisioned that a tool is simply another Kernel process executing on one or more of the processors in the network. As such, the tool has access to all the Kernel primitives. Using these primitives along with the tool interface described below, a number of potential tools could be built, such as:

- Process performance monitor: compiles statistics about the runtime performance of a process(es).
- Processor performance monitor: Compiles processor-level statistics.
- Network performance monitor: Compiles network-level statistics.
- Message performance monitor: Compiles statistics about the frequency of messages, average message length, peak bus usage, etc.

Given the above motivation for the tool interface, the actual form of the interface is driven by the following concepts:

- The tool needs easy access to all the information of the Kernel. Whether or not the tool can make use of the information is not the Kernel's concern. The key is that the Kernel must provide visibility into everything it knows intrinsically, without expending resources to combine that intrinsic knowledge in any way.
- The extraction of information based on what the Kernel knows is left to the tool (and indeed, it is deemed to be the function of a tool). It is in the domain of the tool where the intrinsic Kernel information is combined and presented in some context-specific manner.
- The internal Kernel information must be provided in a manner that does not compromise the integrity of the Kernel; this implies a read-only access to the Kernel's internal data structures.
- The performance impact of using the tool interface must be predictable. Obviously, the performance impact will not be entirely predictable given the non-determinism inherent in the activities being monitored. But the tool interface bounds the impact in a way that gives insight into the potential performance impact of a tool (of course, the tool is a process that can be monitored like any other process in the system, so its performance may be determined empirically). The tool should consume predictable resources generally (not just clock cycles), e.g., storage, message bandwidth.
- An application should *never* have to be modified simply to use a tool (while this may not always be possible, it is nevertheless a desirable goal). Therefore, while some of the information made available via this interface could be acquired by having the tool communicate directly with an application process, this approach is rejected as bad tool design and a distinct detriment to the application software of an embedded system. (Note that the Main Unit is used to establish the process topology and is not considered application code.)

In general there are two classes of Kernel information that may be of interest to a tool:

process information and interrupt information. The primitives defined below describe the information available and the mechanisms provided to access this information.

23.1. Primitives

23.1.1. Process information

The Kernel knows about all processes in the system; therefore, the tool interface provides the following information about processes to the user:

- Process attributes, including:
    - Process identifier
    - State (see Figure 18-1)
    - Time (the time when the above state was entered)
    - Priority
    - Preemption status
    - Alarm status
    - Primitive return status code
- Message attributes, including:
    - Sender's process identifier
    - Receiver's process identifier
    - Time the message was sent or received
    - Message length
    - Message tag
    - Message contents
- Process table, which includes the following information for every declared process:
    - Local process identifier
    - Remote process identifier
    - Process name
    - Device name
    - Local address
    - State (see Figure 18-1)
    - Priority
    - Preemption status

23.1.2. Interrupt information

In a manner analogous to process information, the Kernel knows all there is to know about the interrupts in the system. Therefore, the tool interface provides the following interrupt information to the user:

Interrupt table, which includes the following information for every interrupt:

- Interrupt name
- Interrupt state (enabled or disabled)
- Handler state (bound, unbound, unknown)
- Handler code address
- Handler stack address

23.1.3. Begin collection

This primitive informs the Kernel of:

- What process attribute to collect and for which process to collect it, or
- What message attribute to collect and for which process to collect it.

The Kernel logs the data asynchronously as the state of the process changes:

- Received message bodies are logged as new messages arrive for the process.
- Sent message bodies are logged as the process sends messages.
- Process statistics are logged when the process changes one of its own attributes or when the scheduler changes its process state.
- Message attributes (no bodies) are logged as new messages arrive and leave.

The Kernel logs data by sending a message to the process that requested the collection operation; this process is presumably a part of the tool.

Nonexistent, aborted, or terminated processes for which information is requested are ignored.

This primitive always succeeds.

23.1.4. Cease collection

This primitive disables the collection of the indicated attribute on the indicated process.

This primitive always succeeds.

23.1.5. Read process table

This primitive reads the Kernel's process table.

This primitive always succeeds.

23.1.6. Read interrupt table

This primitive reads the Kernel's interrupt table.

This primitive always succeeds.

23.1.7. Size of process table

This primitive determines the size of the Kernel's process table.

This primitive always succeeds.

23.2. Blocking Primitives

None.

23.3. Status Codes

None.

Appendix A: Glossary

Absolute (time):	A synonym for *epoch* time.
Ada:	ANSI/MIL-STD-1815A. Related information can be found in Section 4.1.
Alarm:	A single timer associated with a process that may expire during process execution. If it does expire, a change of process state occurs, and the exception **alarm_expired** is raised. Related information can be found in Chapter 13 and Chapter 22.
Asynchronous (event):	An event that occurs while the affected process is performing other work or is waiting for the event.
Blocked (process state):	A process that is (temporarily) unable to run. All process states are described in Chapter 18.
Blocking (primitive):	A Kernel primitive that causes the process state to become blocked. The "blocked" process state is described in Chapter 18. Each chapter in Part 3 has a paragraph discussing blocking primitives.
Cyclic (process):	A Kernel process with all the following characteristics: it executes repeatedly; it executes within a user-defined time limit; if it overruns its execution time limit (i.e., its "frame"), then the exception **alarm_expired** is raised.
DARK:	Acronym for the SEI **D**istributed **A**da **R**eal-Time **K**ernel Project.
Dead (process state):	A process that is unable to run again. All process states are described in Chapter 18.
Device:	A hardware entity that can interrupt a processor or that can communicate over the system bus.
Distributed:	Executing on more than one processor in support of a single application.
Duration:	The Ada type *duration*; used to measure *elapsed* time. Related information can be found in Chapter 21 and the *Ada Language Reference Manual*.
Elaboration:	The process by which declarations achieve their effect (such as creating an *object*); this process occurs during program execution. This definition is from the *Ada Language Reference Manual*.
Elapsed (time):	The number of TICKs since the end of the application initialization process. Related information can be found in Chapter 21.
Epoch (time):	The value representing the moment at which the processors began to compute elapsed time. Related information can be found in Chapter 21.
Event:	Something that happens to a process (e.g., the arrival of a message, the arrival of an acknowledgment, being killed by another process).
Exception:	An error situation which may arise during program execution. This definition is from the *Ada Language Reference Manual*.

FIFO:	First in, first out.
Interrupt:	Suspension of a process caused by an event external to that process, and performed in such a way that the process can be resumed. (This external event is also called an interrupt.)
Interrupt handler:	Code automatically invoked in response to the occurrence of an interrupt.
Kernel:	Basic system software to provide facilities for a specific class of applications.
Local:	A process executing on the node invoking the primitive.
NCT:	Network Configuration Table.
Network:	Series of points (nodes, devices, processors) interconnected by communication channels.
Package:	A group of logically related entities, such as *types*, *objects* of those types, and *subprograms* with *parameters* of those types. It is written as a *package declaration* and a *package body*. A package declaration is just a *package specification* followed by a semi-colon. A package is one kind of *program unit*. This definition is from the *Ada Language Reference Manual*.
Package body:	Contains implementations of *subprograms* (and possibly *tasks* as other *packages* that have been specified in the package declaration). This definition is from the *Ada Language Reference Manual*.
Package Calendar:	The Ada package Calendar. Related information can be found in the *Ada Language Reference Manual*.
Package specification:	Has a *visible part*, containing the *declarations* of all entities that can be explicitly used outside the package. It may also have a *private part* containing structural details that complete the specification of the visible entities, but that are irrelevant to the user of the package. This definition is from the *Ada Language Reference Manual*.
Postcondition:	An assertion that must be true after the execution of a statement or program component.
Pragma:	Conveys information to the Ada compiler. This definition is from the *Ada Language Reference Manual*.
Precondition:	An assertion that must be true before the execution of a statement or program component.
Primitive:	Basic Kernel action or datum.
Process (Kernel):	An object of concurrent execution managed by the Kernel outside the knowledge and control of the Ada runtime environment; a schedulable unit of parallel execution. Related information can be found in Chapter 2.
Process stack:	Built by the Kernel when creating a Kernel process. The process stack contains a stack plug (to prevent the propagation of unhandled exceptions), a dummy call frame (pointing to process termination code), and a place for process-local variables.

Processor:	Central processing unit (CPU).
Real-time:	"*When* it is done is as important as *what* is done."
Remote:	A process not executing on the node invoking the primitive.
Runtime:	The period of time during which a program is executing.
Running (process state):	A process that is executing on its processor. All process states are described in Chapter 18.
Semaphore:	A mechanism for controlling process synchronization, often used to implement a solution to the mutual exclusion problem. Related information can be found in Chapters 8 and 17.
Status code:	Generic term used to indicate the status of the execution of a Kernel primitive. A status code may correspond to an output parameter of some discrete type or to an exception. Related information can be found in Chapter 4.5. In addition, each chapter in Part 3 has a paragraph discussing status codes for each primitive.
Suspended (process state):	A process that is able to run but cannot because a process of higher or equal priority is running. All process states are described in Chapter 18.
Synchronous (event):	An event that happens while a process is looking for that event.
System bus:	Communication medium connecting processors and devices into a network.
Task:	An Ada language construct that represents an object of concurrent execution managed by the Ada runtime environment supplied as part of a compiler. Related information can be found in Chapter 2 and the *Ada Language Reference Manual.*
TICK:	The smallest resolvable interval of time. The Kernel references time in units of TICK. Related information can be found in Chapter 21.
Time:	The Ada type *time*; used to measure *epoch* time. Related information can be found in Chapter 21 and the *Ada Language Reference Manual.*

Appendix B: Mapping from Kernel Primitives to Requirements

Deleted - 12 April 1989. This information is now contained in the Ada package specifications.

Appendix C: Mapping from Requirements to Kernel Primitives

Deleted - 12 April 1989. This information is now contained in the Ada package specifications.

Appendix D: Requirement Results

This appendix presents the results of the performance testing for the 68020 version of the Kernel. These results come from measurements completed on 1 September 1989. Please note the following:

1. All values are in microseconds unless otherwise indicated.
2. All known time deductions have been applied to the measured value column; the time deduction for *get_current_count* is 11.18 μs.
3. Times for tests 10.2.1 through 10.2.4 were obtained by taking one-half of the round trip times for the corresponding messages.
4. The time for creating a process includes declaring the process as well.
5. A large message is 1024 bytes; a 0-length message is 1 byte.

One final word, all of these values reflect a Kernel to which no optimizations (either via the compiler or via Kernel code restructuring) have been applied.

Requirement Results

Requirement Number	Required value	Measured value
5.2.1	5%	?
5.2.2	5%	?
5.2.3	100 bytes	?
5.2.4	linear	See KAM & KUM
5.2.5	KAM	See KAM
6.2.1	5.0 secs	516002.82
6.2.2	TBD	#
7.2.1	60	1184.02
7.2.2	30	251.78
7.2.3	deleted	#
7.2.4	20	36.82
7.2.5	100 bytes	?
7.2.6	64 bytes	?
8.2.1	25	?
8.2.2	25	47.58
8.2.3	25	57.84

Requirement Results		
Requirement Number	Required value	Measured value
9.2.1	57	164.88
9.2.2	18	79.54
9.2.3	43	*
9.2.4	33	*
9.2.5	18	245.04
9.2.6	76	111.74
9.2.7	33	*
10.2.1	25	2502.79†
10.2.2	25	43533.20†
10.2.3	50	3892.74†
10.2.4	50	43405.86†
10.2.5	15	187.16
10.2.6	15	198.34
10.2.7	20	359.06
10.2.8	20	343.20
10.2.9	25	431.78
10.2.10	25	1549.32
10.2.11	128 bits	?
11.2.1	15	35.78
11.2.2	10	12.12
11.2.3	20	38.82
11.2.4	16 bytes	?
12.2.1	18	301.96
12.2.2	18	484.68
12.2.3	18	45.18
12.2.4	200	?
12.2.5	TBD	*
13.2.1	10	237.90
13.2.2	10	101.66
13.2.3	TBD	1486.58

Requirement Results		
Requirement Number	Required value	Measured value
14.2.1	10	?
14.2.2	5	?
14.2.3	5	300
14.2.4	Predictable	See KUM
14.2.5	Consistent time	See KUM
14.2.6	Consistent data	See KUM

TABLE KEY:

? - no measurements taken
# - test deleted
* - test requires a logic analyzer to complete
† - no deductions for transmission time applied, test requires
a logic analyzer to complete

References

[ALRM 83] American National Standards Institute, Inc.
Reference Manual for the Ada Programming Language.
Technical Report ANSI/MIL-STD 1815A-1983, ANSI, New York, NY, 1983.

[artewg-interface 86] Ada Runtime Environment Working Group.
A Catalog of Interface Features and Options for the Ada Runtime Environment.
Technical Report Release 1.1, SIGAda, November, 1986.

[artewg-model 86] Ada Runtime Environment Working Group.
A Canonical Model and Taxonomy of Ada Runtime Environments.
Technical Report, SIGAda, November, 1986.

[artewg-survey 86] Ada Runtime Environment Working Group.
First Annual Survey of Mission Critical Application Requirements.
Technical Report Release 1.0, SIGAda, November, 1986.

[Firth 87] Firth, R.
A Pragmatic Approach to Ada Insertion.
In *Proceedings of the International Workshop on Real-Time Ada Issues,* pages 24-26. May, 1987.

[KAM 89] Bamberger, J., T. Coddington, C. Colket, R. Firth, D. Klein, D. Stinchcomb, R. Van Scoy.
Kernel Architecture Manual.
Technical Report CMU/SEI-89-TR-19, ESD-TR-89-27, Software Engineering Institute, December, 1989.

[KFD 89] Bamberger, J., C. Colket, R. Firth, D. Klein, R. Van Scoy.
Kernel Facilities Definition.
Technical Report CMU/SEI-88-TR-16, ESD-TR-88-17, ADA198933, Software Engineering Institute, December, 1989.

[Tanenbaum 81] Tanenbaum, A.S.
Network Protocols.
Computing Surveys 13:453-489, 1981.

[Workshop 88] Ada UK and SIGAda.
Second International Workshop on Real-Time Ada Issues, Devon, UK, ACM Press, 1988.

[Zimmermann 80] Zimmermann, H.
OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection.
IEEE Transactions on Communications COM-28:425-432, 1980.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-88-TR-16	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-88-TR-17
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification)
KERNEL FACILITIES DEFINITION Kernel Version 3.0

12. PERSONAL AUTHOR(S)
Judy Bamberger, Currie Colket, Robert Firth, Daniel Klein, Roger Van Scoy

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ______ TO ______	December 1989	114

16. SUPPLEMENTARY NOTATION
Please note: This document replaces CMU/SEI-88-TR-16, ESD-88-TR-17, "Kernel Facilities Definition" DATED JULY 1988.

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

Ada	Kernel
distributed	real-time
DARK	operating systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document defines the conceptual design of the Kernel by specifying: the underlying models, assumptions, and restrictions that govern the design and implementation of the Kernel; and the behavioral and performance requirements to which the Kernel is built. This document is the requirements and top-level design document for the Kernel.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED, UNLIMITED DISTRIBUTION

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
KARL H. SHINGLER	412 268-7630	SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE